MAC TR-117

AN INPUT/OUTPUT ARCHITECTURE FOR VIRTUAL MEMORY COMPUTER SYSTEMS

David D. Clark

January 1974

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

AN INPUT/OUTPUT ARCHITECTURE FOR VIRTUAL MEMORY COMPUTER SYSTEMS

BY

David Dana Clark

Submitted to the Department of Electrical Engineering on August 20, 1973 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

In many large systems today, input/output is not performed directly by the user, but is done interpretively by the system for him, which causes additional overhead and also restricts the user to whatever algorithms the system has implemented. Many causes contribute to this involvement of the system in user input/output, including the need to enforce protection requirements, the inability to provide adequate response to control signals from devices, and the difficulty of running devices in a virtual environment, especially a virtual memory. The goal of this thesis was the creation of an input/output system which allows the user the freedom of direct access to the device, and which allows the user to build input/output control programs in a simple and understandable manner. This thesis presents a design for an input/output subsystem architecture which, in the context of a segmented, paged, time-shared computer system, allows the user direct access to input/output devices. This thesis proposes a particular architecture, to be used as an example of a class of suitable designs, with the intention that this example serve as a tool in understanding the large number of interactions which exist between the various parts of the input/output system. These interactions make the design of an input/output system more complex, for they prevent the independent investigation of the various input/output system parts. Using this specific system, the thesis draws several conclusions, some of which are 1) that in order to provide a coherent and understandable program structure, input/output operations should be contained in a process dedicated to the task, which uses interprocess communication facilities to signal to other processes, 2) that to allow the user to refer to his device in a simple fashion while using the segment access controls to protect his devices from other users, the input/output device should be interfaced as a number of memory words, which can be mapped into the environment of the user as a segment, 3) that the virtual memory can meet the timing needs of the input/output system without compromising its own functions by the use of time limits on the duration of the input/output operations, and 4) that interrupts should not be part of the user environment, but should be hidden from the programmer, so that the input/output program he provides is sequential rather than interrupt driven in structure, a much preferable form.

THESIS SUPERVISOR: Jerome H. Saltzer
TITLE: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Professor J. H. Saltzer, and my thesis readers, Professor F. J. Corbató and Professor S. Patil, for their advice and guidance. Their suggestions have been most helpful in shaping the ideas in this thesis. The comments of others who have reviewed the material are also gratefully acknowledged.

Without the assistance of Muriel Webber, who typed drafts from countless pages of my handwriting, the thesis might not have proceeded at all.

Finally, I would like to thank all those, and especially my wife Susan, who have given me encouragement and support during this somewhat protracted undertaking.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No N00014-70-A-0362-0006.

TABLE OF CONTENTS

Chapter

LIST OF FIGURES

Chapter 1

Introduction

The last few years have seen a great advance in the sophistication of computer operating systems, particularly with the interface between the user and the computing resource, as certain features of the computer, not optimally structured as far as the user is concerned, are modified by software or hardware to provide a better interface. Examples of these modifications include time-sharing, which adjusts a computer to the speed and response needs of users, and virtual memories, which remove the limitations imposed by the size of the primary memory of the computer.

In contrast to other computer subsystems, the user input/output subsystem has undergone relatively little evolution. Thus, even in a fairly sophisticated operating system, the user wishing to do his own I/O can still discover an awkward and restrictive I/O interface. This state of affairs holds because the insights that form the basis of an orderly system implementation do not exist for I/O. I/O subsystem implementations are still complex and ad hoc, with resulting disadvantages. The purpose of the thesis is to identify the problems which are central to the complexity of I/O subsystem design, and to develop the understanding which will allow orderly solutions to these problems.

In this thesis the term I/O will be used to mean I/O performed at the request of the user, rather than I/O performed by the system to support system functions. For example, input or output to the disk to sup-

port paging will not be considered. Another way of describing the I/O to be studied here is that it is I/O to devices which are controlled by the system only to the extent of granting or denying permission to use them. From the system point of view, the device is just a source or sink of data.

The I/O subsystem will be considered in the context of a large multi-processing time-sharing system. The system will provide multiple virtual memories, each consisting of a paged segmented address space, with protection mechanisms provided to control access separately to each segment in the virtual memory. Each user is provided with one or more processes, each characterized by a current value of an instruction counter and an associated virtual memory. In other words, every process has its own address space. The supervisor will be distributed; that is, the programs which constitute the operating system are implemented as segments containing code which execute in the process of the user on whose behalf they are run. The system code exists in all address spaces, rather than being isolated in an address space of its own.

This particular kind of computer operating system was chosen as being far enough advanced to make this research interesting but well enough developed to provide faith that the implications and interactions of the various features are indeed understood. One implemented system with these features is the Multics system (8,9,31). This system will be used from time to time through the thesis as an example; it has been chosen as an example because of the author's familiarity with it, and because its generality tends to subsume most other systems. It is very

important to have an example such as this, for, in the field of operating system design, the knowledge is lacking which allows the analytical demonstration of the effectiveness of given techniques. Lacking such analytical tools, it is necessary to turn to existence proofs, in the form of implemented systems, to show that some proposal is indeeed practical. It is important to stress, however, that prior knowledge of the Multics system is not required in order to understand this thesis. The observations which the thesis will make about I/O are not restricted to Multics by any means, but are believed to be rather more general.

In an attempt to identify the particular aspects of I/O which will be considered in this thesis, a possible modularization of an I/O subsystem is presented in Figure 1-1. The thesis will concern itself with two modules in the figure: device dependent functions and hardware interface and control, because most complications seem to be centered here, and because proper design of these modules is crucially related to the proper functioning of the operating system as a whole. Little will be said about the design of the I/O device itself, the bottom module in the diagram. Other than assuming that devices come in a wide range of transfer rates and data path widths with various timing constraints, device details will be ignored. The reader may think in terms of disks, tapes, printers, typewriters, etc.

The upper modules in the diagram will be excluded because design of these functions seems much better understood, and because these functions seem less central to the basic supervisor operation. A device-independent interface at a fairly abstract level has been achieved or discussed

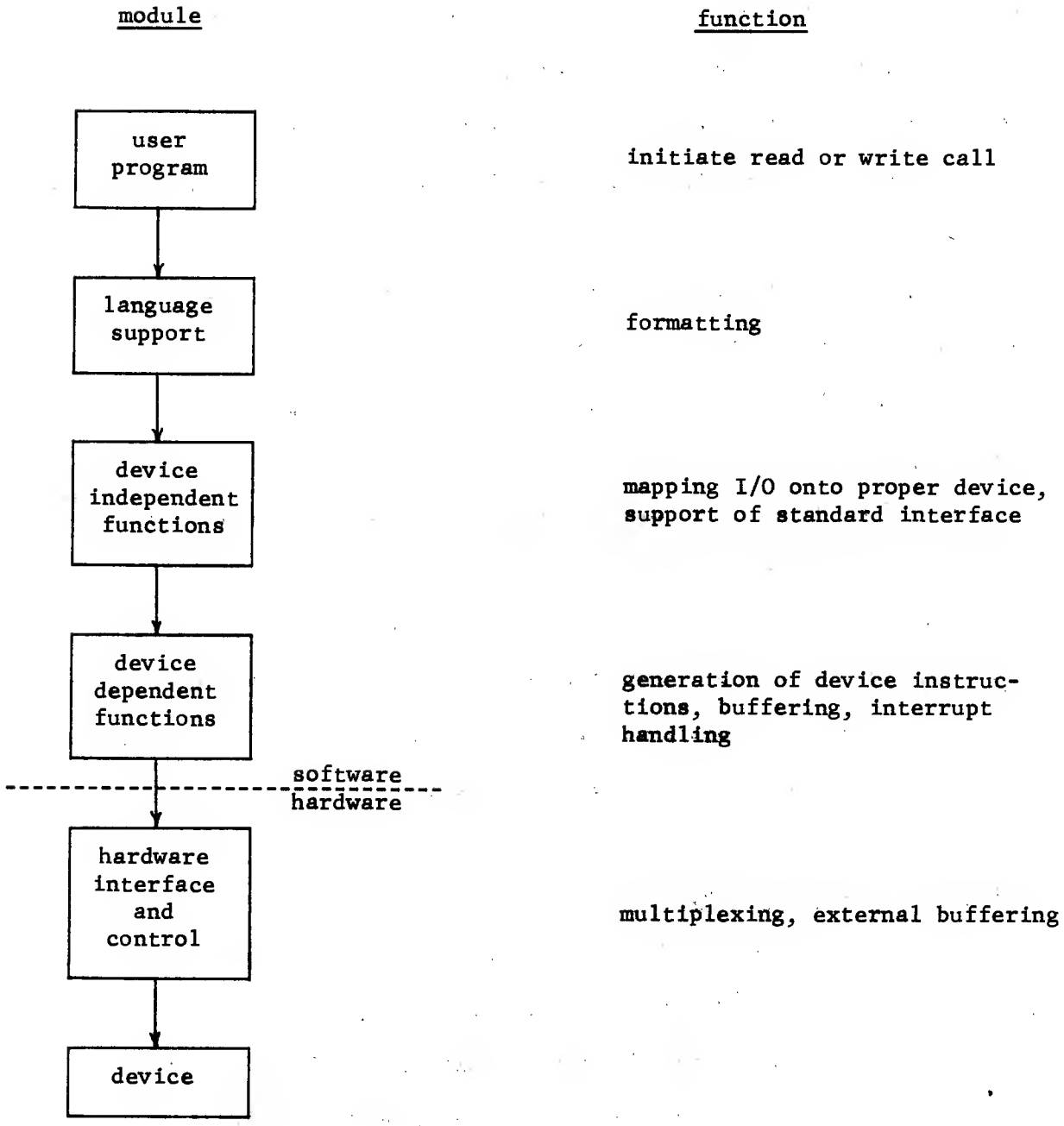

Figure 1-1: Possible modularization of I/O system.

in several forms. Multics, for example, has provided a general interface, in the form of a number of subroutine calls, which allow equivalent operations on different devices to be expressed in the same way. One routine reads, another writes, and so on. To specify which device is to perform the operation, a symbolic name is supplied on each call. Since the relation between name and actual device may be modified dynamically by other calls, the particular device invoked by a given call at this interface may be changed without altering the program making the call. For a more detailed description of this interface, see the paper by Feiertag and Organick (18).

An alternate form of device-independent interface is to model devices as separate processes, and to send information being read or written messages through the interprocess communication mechanisms of the system. This is a slightly more restrictive interface, for only those aspects of the device which are mirrored in the interprocess communication mechanism can be affected at the interface.

Defects of Current I/O Systems

In order to understand what this thesis hopes to achieve, it is first necessary to understand what is wrong with I/O systems as they exist now. Examination of current systems reveals that the various superficial defects observable are caused by five general problems which beset the I/O subsystem. These five problems are as follows:

First, I/O subsystem designers have had a hard time matching the resources actually consumed by an I/O task to the resources required. I/O devices usually operate at a much slower speed than the speed of the

computer to which they are attached. It was thus realized very early that if the processor were used directly to perform I/O, it would be very inefficiently utilized. A variety of techniques have been introduced to attempt to reduce the resources needed for I/O. The problem is that these techniques have, in the process of achieving their goal, introduced other restrictions and inconveniences. For example, to reduce processor costs associated with I/O, channels (special purpose processors) were devised to stand in place of processors. But this introduced a channel programming language, and the complexities of program structure which result from adding a second processing element to the computation. Another technique developed to reduce processor costs was scheduling of the processor using interrupts. But this development introduced the awkward program structure which interrupts can cause.

To reduce memory costs associated with I/O, special buffering schemes were devised. But buffering can cause two problems. First, the user may be required to perform his I/O indirectly through the module managing the special buffer, rather than directly from his own program. Second, any use of buffers introduces the complexity we will call data pipelining. Data pipelining describes any situation in which items do not move directly between device and memory, but rather inhabit intermediate storage on the way, such as memory buffers, hardware buffers, queues in multiplexors, etc. The result of data pipelining is that the instantaneous description of items transferred is very complicated, so that it is often awkward to determine just how far an I/O operation proceeded, for example if it stopped on an error. This pipeline of data must also be flushed on occasion, say if data for a stopped device is

clogging some multiplexed facilty.

To reduce the cost of peripheral components associated with I/O, the components are shared, or multiplexed, among several devices. But this destroys the uniform appearance of the I/O system (various devices are multiplexed in different ways), and may eliminate the possibility of hardware control of access to devices. Clearly, the resources required for I/O should be minimized, but equally clearly, these various drawbacks should be avoided.

Second, I/O designers have had a hard time matching timing constraints of I/O devices to the timing characteristics of a virtual environment. If a user is given direct access to his device, he will expect it to operate in the environment he sees. But if the user's environment includes a virtual memory, then the real time restrictions of the device must be reconciled with the unpredictable delay which such things as page exceptions cause in a virtual memory reference. The solution to this problem often has again been buffers, with the associated problems discussed above.

Third, I/O designers have had a hard time integrating the asynchronous nature of I/O into the process structure. It is appropriate to view I/O as going on in parallel with, or independent from, the rest of the user's computation. The user's computation will proceed faster as a result and system throughput will increase. The desired parallelism can, however, be produced in a variety of ways, some less desirable than others. The use of the interrupt to simulate parallelism is an obvious idea, but it produces a structure with various undesirable characteristics, as will be discussed in the next chapter.

Fourth, I/O designers have had a hard time developing mechanisms which apply to a large class of devices. This is a slightly different sort of problem than the three before. It is perhaps more of a design criterion which has so far been violated. A good example of this problem is buffering, mentioned above as a solution to several problems. If designers could identify the similarities between several devices, and build one buffer manager which served all, they would derive various benefits. Obviously, one benefit would be that there was only one manager to code, install, and maintain. Perhaps more importantly, one buffering strategy could be integrated into the system itself as part of the virtual memory manager, whereas it is much more difficult to justify including a collection of specialized strategies, which will rather exist more like accessories fastened to the outside of the system. The benefits of having the I/O buffer scheme fully integrated into the virtual memory should outweigh any loss of local efficiency which follows from exploiting the similarities rather than the differences between devices.

Fifth and last, the I/O designers have had a hard time devising an I/O architecture which is clean, simple, and elegant. The result of this is that I/O programs are difficult to write, the correct functioning of mechanisms is difficult to prove, and the mechanisms themselves are difficult to understand. This problem, like the previous one, is in the nature of a design criterion. Clearly, this thesis cannot afford to restrict itself to issues of functionality; it must consider issues of elegance and cleanliness as well, for it is crucial that the architecture presented in this thesis be easy to understand as well as functionally correct.

Overview of Thesis

These preceding five observations allow a general statement of the goal of this thesis, which is to show that by making the proper assumptions and the proper design decisions, it is possible to build a system which is compatible with a virtual memory machine of the kind discussed here, and which at the same time succeeds in correcting the five defects discussed above. This is a rather general statement of the thesis goal. It can be stated more specifically as follows: the failure of I/O system designers to cope with the five given defects has two obvious consequences, which this thesis intends to eliminate. The first consequence is that while it is always desirable to have as little of the system as possible within the privileged supervisor, in order to foster flexibility and to reduce the bulk of the code on whose correctness the system depends, the code implementing the various I/O functions often require supervisor privileges and protection, for such purposes as control of multiplexed modules, shared buffers, or the channels themselves. This in turn implies that the user cannot replace these I/O programs, but must be content with what the system provides. Second, programs implementing these functions are difficult to write, error-prone, and very complicated in structure. One classic cause of this complexity is the interrupt, which can cause a very awkward program structure. Other causes mentioned above were special channel languages and improperly implemented parallelism.

The goal of the thesis can now be stated, in terms of these consequences, as attempting to build a system which, first, gives the user direct access to his I/O device, rather than requiring him to use inter-

posed system code, and second, provides an environment in which building an I/O module is not so exceptionally difficult as it now seems to be.

In order to achieve this goal, the thesis must proceed in stages; there is no one insight which will sweep away all difficulties at once. Rather, there are several design decisions which must be made, and, perhaps more difficult, which must be integrated with each other. One of the problems of I/O is that the various issues of I/O design influence each other to a high degree, so that the implementor can easily become lost in a maze of interacting solutions.

The technique this thesis will use to launch an orderly attack on the problem is to make several simplifying assumptions, the effect of which will be to ignore certain of the defects listed above. In particular, a system will first be presumed in which I/O may consume any amount of resource needed, and in which there are no real timing constraints anywhere in the I/O system. Elimination of these two problems will allow initial concentration on the more fundamental issue of what appearance the I/O system shall have in the virtual environment of the user. How shall the parallelism implied by I/O be represented? Perhaps more basically, what shall the representation of the I/O device itself be in the user's environment? The thesis will propose a particular solution to this subset of the problems, and will then demonstrate the validity of this solution by removing the various simplifying assumptions one by one, and evolving a solution which succeeds in coping with the defects thus revealed. As these assumptions are reconsidered, the thesis will proceed from a fairly idealized I/O subsystem to one which might be practical by today's standards.

The thesis will be organized as follows. In the next chapter, the first version of the I/O system will be presented. It will depend on several simplifying assumptions, as mentioned above. The most distinctive and important feature of this simple I/O system is that the I/O device is represented in the environment of the user as some number of words in his virtual address space. This particular device interface has several important advantages: it allows the user to reference his device without using special I/O instructions, it allows the device to be protected from access using those tools which protect segments in the virtual memory, and it causes a great simplification in the role and architecture of channels. The chapter will then discuss how to implement the parallelism appropriate for I/O, and the correct program structure to deal with errors and with signals from the device. It will do so by assuming the existance of several processes, one or more for the main computation of the user, a separate I/O process, and additional processes which wait for errors to occur. The result of this process structure is that no device ever "interrupts" a process asynchronously.

Chapters 3 and 4 deal with the problem of reconciling the timing characteristics of the device and of the virtual memory. Chapter 3 discusses a modification to the virtual memory, in which the I/O process is allowed to fix necessary pages in memory during I/O operations. The chapter shows that imposing an enforced time limit of negotiable duration on this fixing of pages is a sufficient constraint to make the technique acceptable to both I/O process and virtual memory. In this chapter the use of the time limit will restrict the technique to record-oriented devices, but in Chapter 7 the restriction will be relaxed, so that an

interface between I/O and the virtual memory is provided which is applicable to a very wide variety of devices. Chapter 4 discusses an alternative to this technique, in which the virtual memory is not modified, but rather buffers are inserted between the device and the rest of the system. The chapter discusses the problems of buffers, or more generally the problems of data pipelining, which must be resolved before buffers can be utilized. It concludes that while buffers can, under certain circumstances, cause several complications, they can, if properly designed, prove very helpful. The buffer design which Chapter 4 develops can be used for several purposes other than relieving timing conflicts with the virtual memory. In particular, buffers are helpful in the multiplexing of resources, especially processors.

Chapter 5 will introduce various sorts of multiplexing into the I/O system. It will conclude that while most kinds of multiplexing are quite appropriate, there are certain sorts, the multiplexing of I/O ports and the multiplexing of certain kinds of buffers, which are capable of causing trouble. The chapter will identify these and show what problem they have.

Chapter 6 considers how to reduce the cost of processors used to perform I/O. It discusses the techniques mentioned above: channels and scheduling by interrupts. For channels, it shows that representing a device as a sequence of memory words allows a great simplification in channel structure. For scheduling by interrupt, it shows that a structure can be imposed on interrupts which avoids the bad effects interrupts so often cause. The chapter also shows that buffers as developed in Chapter 4 can be used as tools to reduce processor costs.

Chapter 7 considers reducing memory costs. It reconsiders Chapter 3, in which a modification was performed on the virtual memory manager to interface it successfully to I/O, and it extends the class of devices to which the modification can be applied, while reducing cost at the same time. The resulting technique is applicable to essentially all the devices which the thesis will have considered.

Chapter 8 will conclude the thesis by reviewing the total system which results from the combination of these various techniques. By this point in the thesis, many specific issues will have been discussed, including buffering and data pipelining, I/O language semantics and syntax, the I/O device interface, parallelism, multiplexing, and asynchronous virtual memory interaction. Clearly, an integral part of this thesis must be to show the proper role for issues such as these within the I/O system. Equally clearly, these specific issues are only part of the thesis. More important is the combining of all of these issues in such a way that a system results which conforms to the broad goals stated above. Hopefully, one result of the thesis will be to give insight into the relation between the specific issues and the general goals. This is the understanding which is really needed and currently lacking in I/O system design.

Review of Related Work

Research on I/O systems can be divided into two classes: those papers which consider some small portion of the I/O system, and those papers which try to integrate several issues to come up with a coherent overview of the I/O system as a whole. Papers in the former category

are far more common.

Buffering, for example, has been the subject of a great many papers. One of the most common topics is determination of the proper size of a buffer, given a particular buffer strategy, a question which this thesis will largely neglect. It would be hopeless to try to reference all of the queueing theory papers which might bear on this topic. Papers specifically related to computer I/O buffering have been written by Chang (3), Chu (4,5,6,7), Delgalvis (13,14), Dor (16), Gaver (20), and Wolman (40). Some of these papers might be applicable in a practical implementation of this I/O system, but we shall not be concerned with this sort of result in this thesis.

Considerable concern has been given to scheduling of the processor so as to give proper response to I/O tasks. Queueing theory has been employed to attack this problem. Muntz and Coffman (30) attempt to find the minimum execution time of a collection of interrelated tasks given the execution time of the individual tasks. Held and Karp (24) attempt to find the optimum scheduling order given similar conditions. Manchester (27) finds conditions such that starting and finishing time limits are met. These papers are not directly applicable to this thesis, because of the various assumptions which they make about the tasks. In particular, they are more concerned with scheduling a number of interrelated tasks, rather than independent tasks. But they are interesting because of the knowledge which they require of each task. In Chapter 3 of this thesis, tasks will be characterized by a maximum running time. These papers, in general, require two other parameters, the maximum time before which the task must be completed, and the minimum time be-

tween successive requests to run the task. Papers by Fiala (19) and Strollo, Tomlinson, and Fiala (38), have shown that if all real-time tasks are described by these three parameters, it is possible to devise a scheme which will integrate scheduling of these tasks into a time-sharing system of the sort envisioned here. These papers describe an analytical technique to discover whether a given collection of tasks can be run within the constraints of the parameters, and also describe various scheduling rules. The thesis will not discuss guaranteed response time scheduling. It will be assumed that if, in order to make some device work properly, such scheduling is required, then the techniques described in these last two papers could be integrated into the I/O system of this thesis. The thesis will lay sufficient groundwork that such an integration should not be difficult.

One difficulty with I/O is that in order to refer to the device itself it is often necessary to use some specialized language. Several attempts have been made to provide a device representation which could be made part of a high-level language. Gertler (21) describes an addition to Algol which allows an I/O device to be manipulated as a variable. Boulton (2) describes a similar modification to PL/I. The IEEE Transactions on Industrial Electronics and Control Instrumentation, Volume IECI-15, 2, December, 1968, contains papers on a variety of schemes which allow device control programs to be written in Fortran. This thesis will achieve a similar goal of representing the device in a high-level language, but it will do so in a form somewhat different from that above, in that our high-level language representation will directly mirror the hardware representation of the device, which was not a goal in

the above papers.

The next chapter will develop a rather idealized I/O system, which will gain its considerable simplicity by ignoring two objectives, the controlling of resource usage and the meeting of the device time constraints. The reader may feel that the simplicity so achieved is deceptive, in the sense that there are probably other assumptions made which would render even this system complicated in practice. As an indication of the simplicity which can in practice result, the paper by Hatch (23) is interesting. It describes an implemented system which disposes of these two objectives, first by the use of channels, and second by keeping all of the user's storage in core at all times. As may be imagined, this imposes some other limits on the user, but the I/O system which results is rather simple and clean. Even with limited hardware support, the user may write and execute his own channel program with minimal system intervention.

One paper which directly considers the integration of I/O into a paged, segmented, virtual memory system is the thesis by Smith (37). This thesis, however, deals with only two topics in particular. One topic is the structure which the I/O control program should have, given that the device is controlled by a channel rather than the processor itself. The thesis concludes that the I/O control program is best structured as a single sequential process which moves itself explicitly from processor to channel and back as necessary. This is a result with which we agree in principle; it will be discussed in Chapter 6. The other topic considered by Smith is the architecture of the associative memory which would be used in the conversion of virtual to real addresses.

This thesis will not consider in detail the utilization of such an associative memory. One point, however, is that Smith fails to consider the need to clear the associative memory, which causes problems because it places a large transient load on the address conversion machinery. It is not as obvious as Smith would suggest that all channels should take advantage of an associative memory. The most important problem with Smith's thesis is that he fails to integrate into his scheme the important issues of memory management and asynchronous virtual memory interaction. He presumes that any page which will be needed is already in primary memory, without discussing how it got there or the cost of keeping it there.

Wirth (41) attempts to deal with the issue of parallelism in I/O. The important conclusion he reaches is that while parallelism is an appropriate tool in doing I/O, there are good and bad ways of producing this parallelism. The technique Wirth uses, which is similar in structure to that of this thesis, is to make the I/O program part of a distinct process, which communicates with the main computation of the user by means of the normal system interprocess communication tools. However, the particular technique Wirth uses to structure the I/O process does not give the user full direct access to the device, and restricts the techniques available for error recovery. Also, Wirth does not consider I/O in a virtual memory context, and thus does not consider issues of memory management.

As the above suggests, any orderly I/O system implies some interprocess communication tools. Two well known sets of tools have been developed: Saltzer (35) describes the primitive block and wakeup.

Dijkstra (15) describes the primitive p and v. Either can be made to work. The book by Organick (31) describes the way Multics implements interprocess communication using block and wakeup.

The single feature which most shapes the I/O system of this thesis is that the device is represented in the virtual environment of the user as a sequence of memory words. This is not the normal interface for a device, but there are two computers which have implemented such an interface in some fashion. One is the PDP-11, manufactured by the Digital Equipment Corporation (11,12). The interface to each device in this machine is as a number of memory words, representing data, state, and control information in a fashion similar to this thesis. In other respects the two systems are rather different. The PDP-11, to the extent it has a virtual memory, has not exploited it to control devices. Nor does the PDP-11 use buffers or channels in the novel way which is allowed by interfacing devices as memory words.

A system which more closely resembles the one developed in this thesis is the Plessey 250 system, a large multiprocessor, virtual memory time-sharing system described in several papers (10,17,22,33). The Plessey system is similar to the system of this thesis in that the representation of the device in the virtual address space is as a segment, protected by the system access control mechanisms, and in that buffers are used to eliminate channels, with the processor itself doing the I/O. The most important difference between this system and the Plessey system is that the Plessey system does not have as a goal direct user access to the device. The papers available do not give great information on this point, but the system implements a possibly restrictive scheduling

strategy, and uses multiplexed buffers in a way which surely prohibits direct access to the devices using that buffer. No information is available discussing Plessey's solutions to the pipelining and synchronization problems raised by buffers, or discussing their views of process structure, error recovery, and related topics. But it is a very important system, because it is the closest system existing to the one being proposed here, and demonstrates the practicality of certain ideas in this thesis, in particular the representation of the device as a segment, and the use of buffers as a processor scheduling tool.

Chapter 2

The Basic I/O System

The purpose of this chapter is to propose a preliminary version of the I/O system, which the rest of the thesis will then develop. This first system will be rather idealized, for as the first chapter explained, two problems will be ignored in its design: the problem that I/O must not consume excessive resources, and the problem that the real timing constraints of devices must be reconciled with the variable timing of the virtual memory. While the system will be in this fashion rather idealized, it will meet the goals of direct user access to the device and elimination of certain causes which make I/O code difficult to write, understand, and debug. The system will also attempt to comply with the design goals of simplicity and generality. The later chapters will show that the achievement of these goals is not compromised when the problems here ignored are taken into account.

Preliminary Simplifications

The first topic of the chapter will be to show in what fashion the thesis will exploit the decision to ignore temporarily the two problems mentioned above. Let us begin by considering a basic characteristic of the I/O subsystem. Computer system modules can be divided into active and passive modules: active such as processors, passive such as memory. I/O contains both aspects: the passive part is the stored data, on tape or disk or in the programmer's head, the active part executes the accessing algorithm to move this data in and out of the passive I/O storage. The characteristic which distinguishes I/O storage from other memory is that

this accessing algorithm, the active part of I/O, can only be of certain forms. I/O storage is not random access; it cannot be addressed to an arbitrary item, but only to a group of items, often called a record, and generally a sequence of items must be transferred starting at a record boundary, so that the accessing algorithm is a sequence of data transfers.

The active aspect of I/O, which implements the accessing algorithm, was implemented in early computers and in simple computers today by the central processor itself. In the more complex systems of today, however, the accessing algorithm is often implemented by a specialized piece of hardware called a channel, or I/O controller, taking advantage of the restricted nature of the accessing algorithm.

Channels contribute to the efficient use of the central processor, but confuse the programming, for invariably the restrictions of the channel prevent the entire accessing algorithm from running on it, so that for parts of the algorithm the programmer must move to the central processor, perhaps by means of interrupts and interrupt handlers. Ideally the programmer should not have to cope with this switching from processor to processor; such switching is an implementation feature due to issues of economy.

Here, clearly, is a chance to take advantage of the fact that within this chapter we are not concerned with issues of efficiency. In order to make the construction of the accessing algorithm as simple as possible, it will be assumed that processors are inexpensive enough that one can be allocated full time to any process doing I/O, and that all I/O will be done by this processor, rather than by some specialized I/O controller. By assuming that the processor is cheap enough to be dedicated to a process

doing I/O, all questions of scheduling processors during I/O are avoided. The processor will just wait for any pending I/O operations to complete.

Just as specialized I/O controllers are often employed to make more efficient use of processors, buffers are often used to achieve more efficient use of memory. Again this chapter will ignore resource consumption and presume that memory is cheap enough that any page needed for the user's I/O may be kept in memory without special restrictions. In addition, since in this chapter the timing constraints of devices are to be ignored, the chapter need postulate no special mechanism to bring pages for I/O into memory. If during I/O processing a page exception occurs because some page is missing, the I/O device is assumed to be able to pause while the page is fetched into memory by the normal means.

The result of ignoring resource consumption and timing constraints, then, is a system in which the accessing algorithm runs directly on the main processor, and in which the algorithm, for all its storage needs, uses the normal memory provided in the user environment.

The Representation of the Device

The previous section has outlined some of the features this I/O system will have; this section will deal with perhaps the single most important characteristic of the I/O system: the representation of the device itself in the environment of the user. That is, given that the active aspect of I/O is represented by the central processor itself, how shall the interface to the passive part be constructed? There must exist some port on the processor to which devices are attached. We must consider what the nature of this port shall be, and what instructions shall be provided to reference

it. Current computers which execute the accessing algorithm directly usually have some specialized port to an I/O bus of some sort, with special instructions to reference it, but in this thesis an alternative will be chosen in which the processor's memory interface is used for I/O devices as well as memory, so that to read or write on an I/O device, the program issues a memory fetch or memory store instruction to a particular address, which the hardware associates with the device rather than a memory word. Such a device interface has been used in two computers, the PDP-11 and the Plessey 250, which are discussed in Chapter 1.

The advantages of this interface are several. First, no modifications are required to the central processor. No special port is needed, nor special instructions, so that the programmer can use any memory fetch or store instruction to reference I/O. This ability means no new language need be learned. Also, as will be demonstrated, the mechanisms which manage and control the segmented virtual memory can be used quite naturally to regulate access to I/O devices.

To see how this interface might work, consider the usual method for interconnecting processors and memories in a multi-processor system. Normally the memory will be implemented as several memory boxes, each holding a fixed number of memory words, with each processor connected to all of the memory boxes. The processor contains a mechanism which, on each memory reference, takes the memory address and directs the reference to the memory box which contains this address.

Given this architecture, it is easy to specify that certain of the addresses be associated with an I/O device rather than with words of memory. One could just replace a memory box with a device, suitably interfaced, but

this would associate with the device all of the large number of addresses normally implemented in one memory box. Better is to provide a module, to be called the device selector, which takes the place of one memory box, and which divides up the addresses associated with the replaced memory box among a number of devices which are connected to the device selector.

The physical arrangement of modules which results from this scheme is depicted in Figure 2-1. In place of one of the memory boxes is a device selector, with the devices in the system connected to it. For purposes of comparison, Figure 2-2 depicts the architecture of a more traditional system, which uses an I/O controller, or specialized I/O processor, to execute the accessing algorithm. The principle difference is that traditionally the devices are connected to the I/O controller itself. Thus, in contrast to the traditional case, the architecture of this thesis separates the active part of the I/O system, represented by the processor, from the passive part of the I/O system, embodied in the device selector. One obvious advantage of this separation is that the total system is more reliable, since any processor, rather than just one in particular, can control any device. Thus the failure of one processor does not disable devices. Other advantages of the separation will be discussed later in the thesis.

Figure 2-3 depicts the relation between the physical devices and the address range. The memory has been implemented as a series of memory boxes, each of which contains 2^k words of memory. In order to implement device attachment, one of the memory boxes has been replaced by a device selector, which in turn takes the 2^k addresses assigned to it and subdivided these into blocks of length n, which it associates with individual devices.

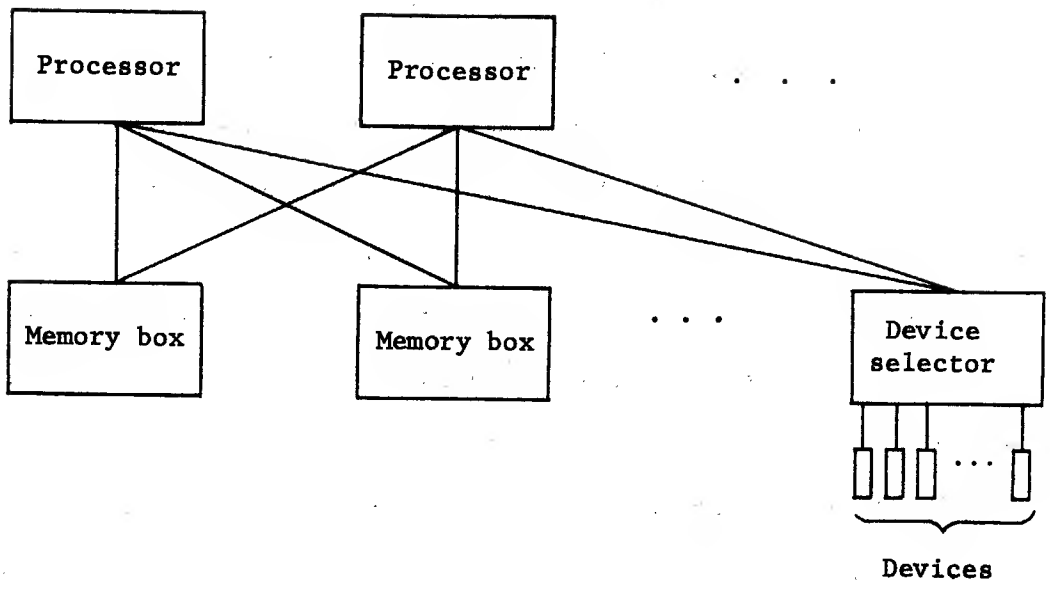

Figure 2-1: Module interconnections with devices represented as memory.

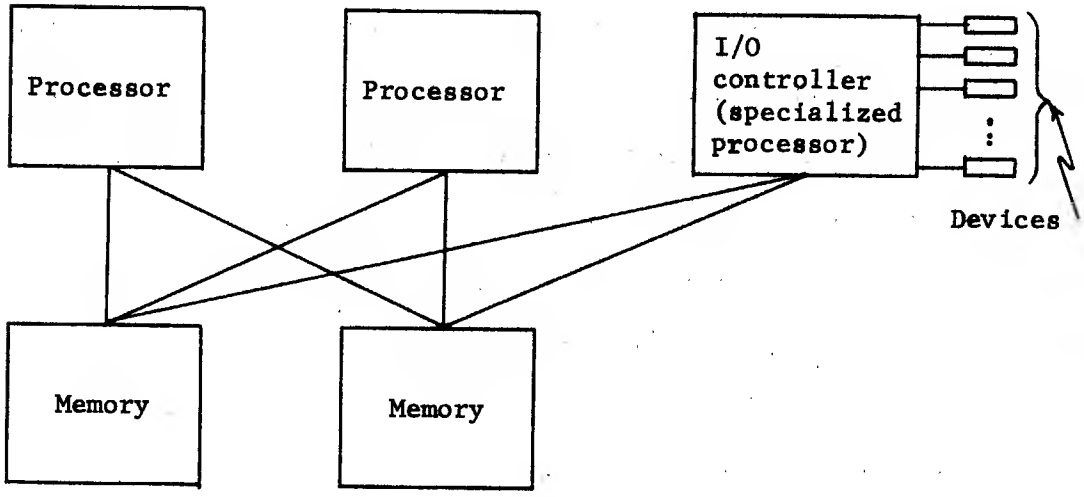

Figure 2-2: Module interconnections in system with specialized I/O processor.

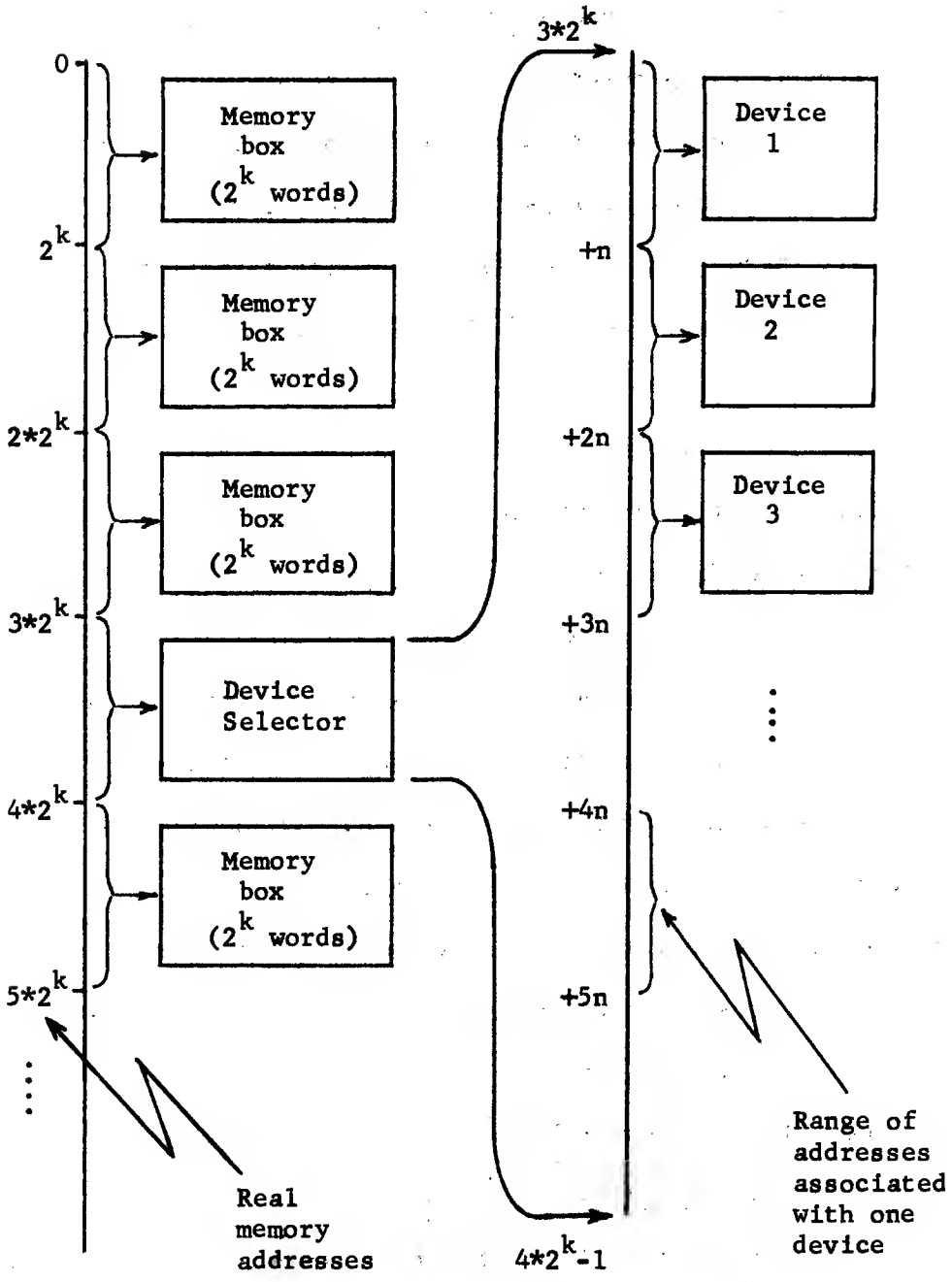

Figure 2-3: Typical memory implementation, showing relation between addresses and physical modules.

The interface as described allows the program to read and write data to the device by repeated reference to a particular memory address. This is not a sufficient interface to allow full control of the device, however, for control information as well as data must be passed to and from the device. For this reason not one but several addresses will be associated with each device. (See Figure 2-3.) These additional addresses will be used as follows.

One of the additional addresses will be used to allow the program to read and write the state word of the device. The state word of the device contains information about the current condition of the device. It will reflect the setting of hardware switches on the device, and the occurrences of errors. By loading it the programmer can alter the state of the device. For example, in a device connected to a modem, the state word will contain bits related to the state of the modem, so that by loading the state word, the modem can be made to hang up, or wait for a call, or answer, and so on. The details of the state word will be ignored in this thesis, but it is assumed that any necessary modification to the state of a device can be achieved by setting bits in its state word.

One aspect of device control is important enough to warrant a special address. This is the record number. The record number is, for devices with records, the number of the current record being accessed. Assigning a value to the address causes the device to position itself at the beginning of the specified record. (This implies that assigning to the record number the value which it contains may cause an action; to wit, backing up to the start of the current record.)

It is claimed that the interface now described, with the three aspects of the device: data, state word, and record address, is sufficient to allow general control of the device at the detailed level.

Mapping the Device into the User's Environment

The last section described a way of representing devices in the real address space of the computer. The user, however, does not see real but rather virtual addresses. This section discusses how and to what advantage the device may be mapped into the virtual address space of the user.

The virtual memory is assumed to be segmented, with the user potentially having a very large number of segments. A virtual address is composed of two parts, a segment number and an offset within the segment. Conversion of virtual addresses to real addresses is performed using a segment descriptor table associated with each address space. The table is indexed by segment number and gives, for each segment, the real starting address and the length of the segment. (If segments are implemented by paging, the real starting address will be that of a page descriptor table in which the real starting address is found, but this detail is irrelevant here.) To the real starting address is added the offset part of the virtual address to find the desired word in real memory.

Examination will reveal that there are certain similarities between devices and segments. Since segments are the basic tool of organization, with each segment expected to hold one informational entity, such as a single program, it follows that protection controls are applied on a per-segment basis. Similarly, access to the I/O system should be granted or

denied on a per-device basis. Also, both devices and segments are manipulated by the user: named, obtained, discarded, etc.

In view of these similarities, devices as well as segments will be identified by segment numbers in the user's virtual memory address space: the segment descriptor table entry corresponding to a device will be constructed in such a way that references to that "segment" will be directed to the real addresses associated with that device. For example, with reference to Figure 2-3, if device 3 were to be added to the address space of a user as segment number d, then the dth entry in the user's segment descriptor table would be filled in with a real starting address of $3 \times 2^k + 2n$ and a length of n. The various offsets in segment d would then map into the various aspects of the device: data, state word and record number.

The result of mapping devices into the user's environment in this way is that the user has access only to those devices which are mapped into his address space. Being able to restrict the user in this way is crucial to the goal of allowing the user direct access to his devices.

A very important advantage of representing the device to the user as a segment is that he can refer to his device in any programming language which lets him refer to a segment. In other words, he can write his I/O control program using an appropriately structured high-level language. Figure 2-4 is an example of an I/O control program written in PL/I, which, first, shows the possibility of referencing the device from a high-level language and which, second, depicts the sequence of actions a user would go through to use a device in the context of this interface.

```
tape_read:procedure(reel_name,where_to_read,num_recs);

/* This is a program in PL/1 to read a tape. The tape identi-
fied by the name "reel_name" will be read into the array
"where_to_read", with "num_recs" records being read. For this
simple example we will assume that no errors will occur.  */

declare reel_name char(*),      /* name of tape */
        where_to_read (*),      /* array into which to read */
        num_recs fixed;         /* how many records to read */

declare tape_addr pointer;      /* device memory address */

declare 1 tape_drive            /* structure of i/o device */
          based (tape_addr),
          2 data aligned,       /* offset of data in device */
          2 record aligned,     /* offset of record address */
          2 state aligned;      /* offset of state info  */

declare rec_size fixed initial (256); /* some number of words
                                         per record */

declare ( recno,wordno ) fixed;         /* indexing */

declare get_tape external entry(char(*)) returns (pointer);
/* This routine will verify the user's access to the tape,
have it mounted, and associate the segment which represents
the device in memory with the structure "tape_drive".   */

tape_addr = get_tape(reel_name);        /* get a drive */
tape_drive.record = 1;          /* position tape at start */

do recno = 1 to num_recs;
  do wordno = 1 to rec_size;
    where_to_read((recno-1)*rec_size+wordno) = tape_drive.data;
  end;
end;

/* No explicit assignment to tape_drive.record is necessary in
the loop because it is assumed that the tape advances to the
next record automatically. */

return;
end tape_read;
```

Figure 2-4: Program in PL/I to read data from a tape.

The first action of the user must be to have the device made accessable to him. For this purpose he must call on the supervisor, which will create a segment in his address space corresponding to the device. In this example the supervisor call is represented by the function get_tape. The PL/I language contains no clear construct by which the programmer may associate a name in the program (e.g., the structure tape_drive) with some object in the environment of the program (e.g., the segment representing the device). In this sense PL/I is deficient in its ability to refer to its environment; and thus to take advantage directly of a segmented address space. In this example, the association is made using the variable "tape_addr", which "get_tape" sets.

Once the segment is accessible to the PL/I program, the user can reference his device directly. The user must first position the tape drive to the first record, by assignment to "tape_drive.record". In a practical case he might also have to set the device in the proper state by assignment to "tape_drive.state". After these preliminaries, the user reads items from the tape into the array "where_to_read" by repeated reference to "tape_drive.data". In general, the user could read, write, and reposition the device as necessary. Eventually, after the user is finished with the device he should call the system and relinquish it.

Hopefully, this example will convince the reader that, ignoring for now the issues of resource consumption, timing considerations and errors, the system as so far constructed allows the user to construct I/O programs in a simple and orderly fashion.

Connection of Device Selector to Devices

The device selector has the responsibility of connecting devices to the system and allowing those devices to be referenced as if they were memory words. The purpose of this section is to show that the representation of the device as memory does not require that the interface between the device and the device selector be very different from interfaces used for devices on systems today. A particular interface will be proposed, which will be used in subsequent chapters.

In an earlier section, three aspects of a device were identified, in particular data, state word, and record number. Normally, at the device interface these three aspects are not represented by three different information pathways, but rather all are transmitted over one path, with additional control lines used to indicate whether the value being transmitted is data, state, or address. The interface described here will take this approach.

Since it was assumed that the processor will wait for the device, if the device is slow, and that the device will wait for the processor, if the processor takes a page fault, the interface must be totally asynchronous. To move information across an asynchronous interface, a technique will be used which involves two signalling lines, the ready line and the acknowledge line. Whenever device or device selector has information to transmit to the other, it will place this information on the appropriate lines, and then signal across the interface on a ready line, which indicates to the receiver of the information that the information is available. The sender then waits until the receiver signals back over the acknowledge line, indicating that

the information has been received. Using this protocol, either side may force the other to wait as necessary.*

From these various observations, a picture of the device interface evolves, as follows. There will be one set of data lines for the parallel transfer of a word of data, state word, or address. Since information can flow in either direction over these lines, two sets of ready-acknowledge lines will be provided, one each way. There will be a set of command lines, used to distinguish data, state word, and address, and to indicate the direction of data flow. These command lines carry information from device selector to device, and have their own set of ready-acknowledge lines. A command over the command lines will be issued by the device selector to the device as part of each transfer, indicating what information is to flow. The device selector will use two pieces of information in generating the command: first, the particular address which was referenced, and second, whether the memory reference was a read or a write request.

For example, to read the next item from a device, the user will issue an instruction which reads from memory, with an address containing the segment number of the device and the offset value for data. The address conversion logic will transform this address to a real one, which will be sent to the device selector. The device selector, noting the particular address and that the instruction is attempting to read, will make up a command to read data, and, placing it on the command lines for the selected device, signal over the ready line associated with the command lines; then wait for

---

* The receiver could dispense with the ready line and presume that the information was available whenever it detected a signal on the information line. If there are several information lines in parallel, however, differences in timing on the lines may cause this technique of detecting information to be error-prone. Thus the use of a single ready line to announce that the information lines have stabilized and may be read.

the device to pick up the command and signal back over the appropriate acknowledge line. After this, since the command was to read, the next step is up to the device, which must place the word to be read on the data lines, and then signal over the appropriate ready line. The device selector will pick up the word, hand it on to the central processor and signal back completion to the device over the associated acknowledge line. Thus a command is issued to the device for every data transfer. Writing data would be similar to reading, except that the selector rather than the device would initiate the data transfer.

A representation of this interface is pictured in Figure 2-5, which identifies each line, and shows for each the direction of signal flow. As the diagram suggests, the interface will be augmented later in the thesis with four more lines. The complete interface is reviewed in Appendix B, which also compared the interface with several others in use today.

The I/O Environment - A Summary

The system so far developed gives the user direct access to each of his devices, while preventing him access to any other device. Since a device is represented as a segment, it can be easily manipulated, using memory fetch-store instructions of the machine, in any high-level language which allows access to segments. Since the user has direct access to the state word of the device, he has very general control over the device, so the user is not restricted in the algorithms he constructs to control his device. The only time he need call on the system supervisor as part of doing I/O is to have the device assigned to him and mapped into his address space in the first place. Thus the system does indeed give the user direct access to his device.

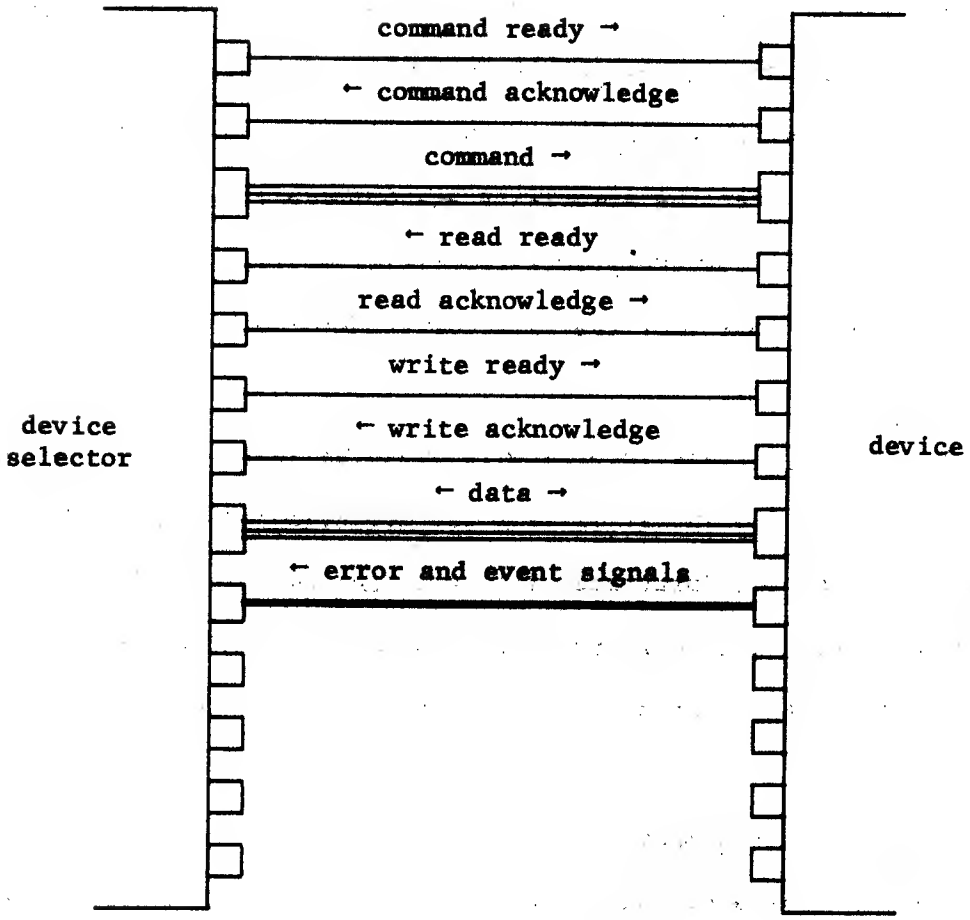

Figure 2-5: Interface between device and device selector.

Parallelism in I/O

One of the three problems introduced in the first chapter was to take proper advantage of parallelism in I/O. The purpose of this section is to determine where parallelism is and is not appropriate in an I/O system design. If the reader will refer to the PL/I example of Figure 2-4, he will note that within that program there is no parallelism of any sort. Quite the reverse, it is completely sequential. This reflects the fact that a single device is sequential in nature, and can do but one thing at a time. Thus, given the assumption that I/O is performed by the main processor, there is no use for parallelism in the construction of a control program for a particular device.

There is, however, some use for parallelism in I/O. If the user can perform his I/O in parallel with other parts of his computation, his total computation will complete sooner, so the system will be more responsive. The system as well as the user will benefit, for having several tasks in parallel means that the system has several rather than one task to schedule. The system, by choosing among these, can better keep all of its resources busy from moment to moment. Further, as the next sections will show, the user can impose an orderly and coherent structure on his I/O task by viewing it as running in parallel with his other computations. Thus, the proper role of parallelism is not in the control of a particular device, but in relating the operation of one device to the other tasks of the user.

How is this parallelism to be produced? There are basically two techniques. One, a fairly common one, is the use of interrupts to multiplex a process between two tasks. The I/O system of this thesis, as so far developed, has no interrupts, so this technique is not relevant. It is

important, however, that this technique be understood, for the thesis will argue that it is an undesirable technique because it confounds the goals of ease of programming and simplicity of structure in the I/O system. Indeed, this thesis will argue even more strongly that the user should never see an interrupt in the traditional sense, but rather that in all cases a signal coming from a device should be intercepted by the system and mapped into some specific mechanism which represents the intent of the signal.

In order to justify this assertion about interrupts, it is necessary to have an example of a system which does use interrupts. For this purpose an alternative architecture will be quickly developed. Imagine a system in which channels exist. For the purpose of this discussion, a channel is just a processor which is specialized to perform certain parts of the I/O accessing algorithm. For example, it might perform the actual transfer of the data represented in Figure 2-4 by the nested do-loops. When the channel has finished its task, it sends an interrupt to the main processor, to indicate that it is done.

In this alternative architecture, interrupts can be used to produce parallelism in the following fashion. Presume that the user has one process, that is, one environment. The effect of the interrupt will be to divert this process from its usual task to a section of code, the interrupt handler, whose function is to determine the cause of the interrupt, and take the necessary steps to get the channel started on its next task. The control will then be restored to the main computation at the point of the interruption. Put another way, the effect of the interrupt is to produce parallelism by multiplexing the user's process between two control points.

The drawback to this view is that since fragments of code from the interrupt handler are executed at arbitrary points during the execution of the other control point, it becomes very difficult to analyze, predict, or reproduce the actual computation performed by the process. The computation is certainly not represented by the user's programs as written. Further, since the two control points are part of the same environment, the degree of interaction is unrestricted. Thus the individual programmer is responsible for designing the means to regulate the interaction. Great skill is needed to device error-free algorithms to synchronize parallelism in this case.

A much better structure, from the programmer's point of view, would be to consider the main process of the user to execute only the main computation, and to run in parallel with a separate process running the I/O control program, to be called the I/O process. There are three advantages of this structure, the first of which is that co-ordination between main process and I/O process can be implemented in terms of whatever interprocess communication mechanisms the operating system supports. This reduces the difficulty of building properly co-ordinated parallelism, since several interprocess communication mechanisms have been developed which allow a logical analysis of parallel structure. Two such schemes are semaphores with the operation p and v, described by Dijkstra (15), and the primitives block and wakeup described by Saltzer (35).

The second advantage of this two process structure is that the main process is not being multiplexed, so the algorithm the main process executes does correspond to the programs as written rather than having the

programs interspersed at random points with transfers to I/O code having unconstrained effect. This makes the algorithm of the main process easier to debug.

The third advantage of separate processes has to do with the structure of the I/O task itself. The algorithm of Figure 2-4 displays, in its written form, the sequential nature of its algorithm. Consider, in contrast, the form which the written program would have if it had been coded as an interrupt handler.

The best way to introduce this "interrupt-handler" structure is by an example of an algorithm to read from a disk a series of records. For each record the algorithm must first seek to the correct address and then read the record. It will repeat this sequence until there are no more records to be read. If channels and interrupts do not exist, this algorithm has the very simple flow chart of Figure 2-6, which would also model the algorithm of Figure 2-4 except for the addition of the seek action.

If the I/O system were implemented using channels, it would be natural for the channels to implement the boxes labeled "seek" and "read". Given that the channel is equipped with an interrupt line over which to signal completion of the current task, some portion of the algorithm must have the responsibility for receiving the interrupt and determining its cause. Including this new portion of the algorithm produces the flow chart of Figure 2-7.

A description of this program structure would be that at each interrupt the control returns to the head of the program, so that what is conceptually a sequential set of operations appears to be alternative paths through the program. The disadvantages of this structure are apparent.

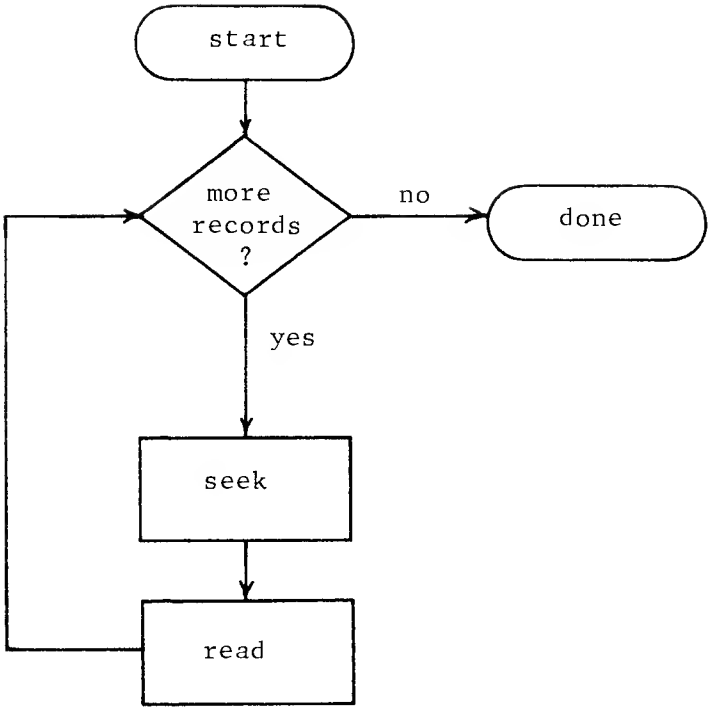

Figure 2-6: Sequential form of flow chart for I/O control program.

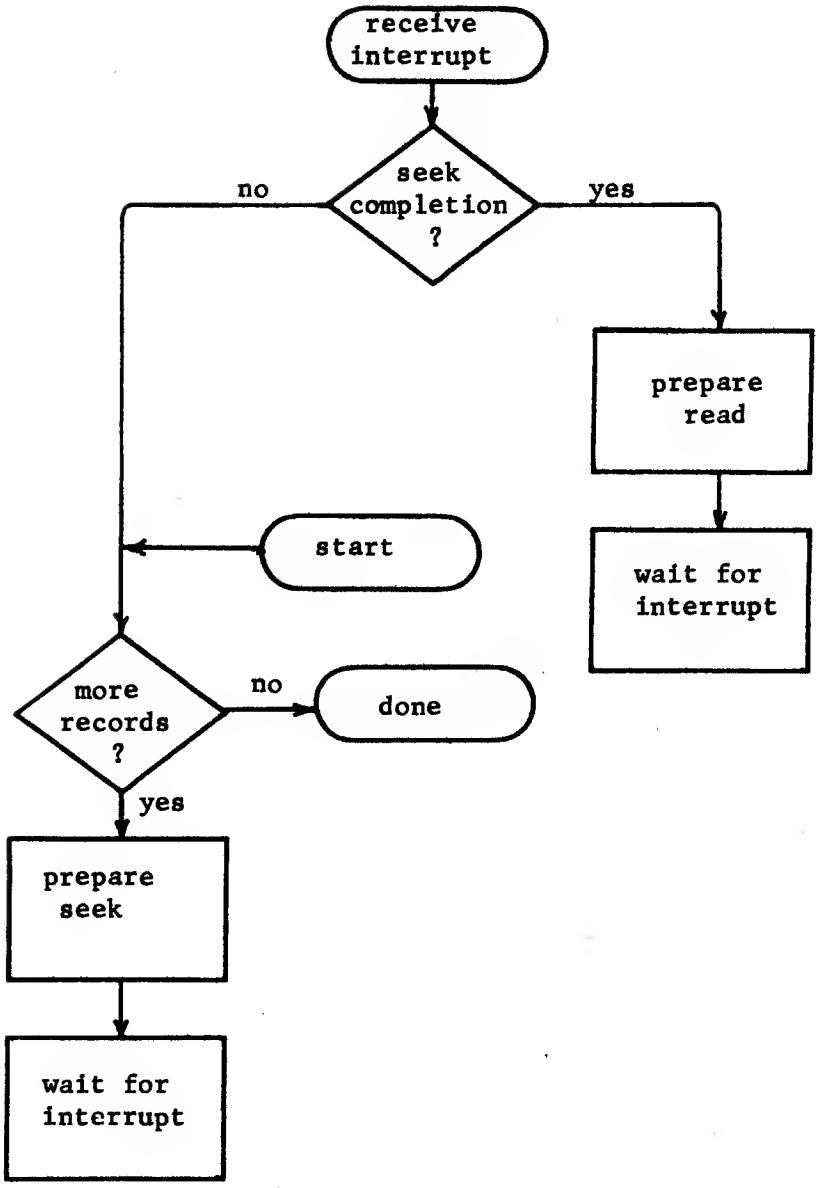

Figure 2-7: Interrupt-driven form of flow chart for I/O control program.

It is no longer obvious from the form of the flow chart that the program contains a loop, nor is it obvious that seeks and reads always come in pairs. The "start" and "done" points occur at an unobvious point in the middle of the program. In a more complex program the results can be chaotic.

The flow chart of Figure 2-7 could be drawn so that it would mimic as much as possible the form of Figure 2-6. Such a flow chart has been pictured in Figure 2-8. While a program with the form of Figure 2-8 would have many of the desirable characteristics claimed for Figure 2-7, it is not obvious that a programmer working with interrupts would create a program with the structure of Figure 2-8 unless the system assisted him, perhaps by providing that portion of the algorithm which received and sorted out the interrupt. Chapter 6 contains a description of how the system might provide this function.

These three advantages of the two process scheme, that the main process is not arbitrarily interrupted, that the sequential nature of the I/O task is not obscured by the interrupt handler structure, and that the interaction between the two tasks can be achieved using formalized interprocess communication tools, are the basis for the decision to use the two-process architecture for the remainder of the thesis. Clearly, the advantage of the two process scheme is not an increase in capability. Quite the opposite, the necessity of working within the framework of processes and interprocess communications might seem rather restrictive to one accustomed to interrupt handlers. But this restriction is in fact what is desired. The justification for the imposition of this separate process structure on the

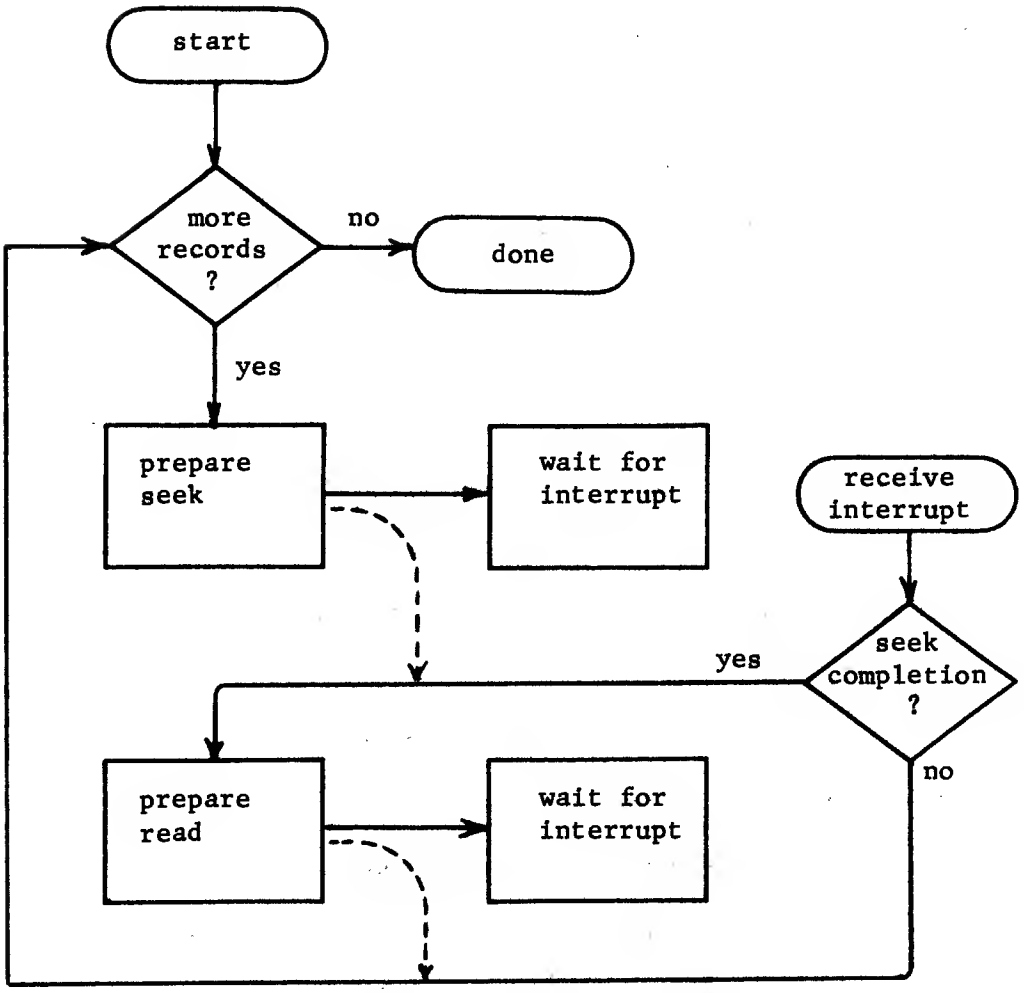

Figure 2-8: Redrawing of Figure 2-7 to resemble Figure 2-6.

I/O task is that the goals of simplicity of design and ease of coding and analysis are thereby fostered, and this is a major goal of the thesis.

While the multiple process architecture nicely complements the I/O device interface developed in the first part of this chapter, it is, in a sense, orthogonal to it. It is not necessary, in order to view the I/O task as a separate process, that the I/O device be modelled as memory words, or that the accessing algorithm run on the processor itself.

The Handling of Errors in I/O

One of the aspects of I/O which cannot be ignored is that I/O is prone to errors. The data storage and data transmission media outside the central processor are constructed using technology which allow occasional transient errors, causing data to be lost or altered. The I/O system must provide a reasonable response to this sort of error, as well as to the obvious programming errors such as the use of an invalid record number.

This section will show that in order to deal properly with errors, it is appropriate to separate them into two categories: those which are triggered by some particular action on the part of the I/O control program, to be called synchronous errors, and those which occur randomly, to be called asynchronous errors. Examples of the former would include a parity error, or an attempt to reference a non-existant record; examples of the latter would be a power failure on a peripheral device or the unexpected disconnection of a communication line. The difficulty with asynchronous errors is that in order for the I/O process to deal with them as they occur, it is necessary to respond at unpredictable times, and this sort of interruption is, as the previous section argued, undesirable.

In order to avoid diverting the I/O process when asynchronous errors occur, a strategy will be employed in which an additional process is used, whose function is to wait for such errors. This additional process can, on detecting an error, perform whatever actions are necessary. The scheme thus avoids the necessity of interrupting the I/O process to deal with asynchronous errors.

The simple case of synchronous errors will be considered first. To begin, how can an I/O process detect that an error has occurred in a device? A mechanism does already exist which can be used for this purpose. A bit in the state word of the device can be associated with each error, and the device can set this bit if the error occurs. The program can then test the state word whenever it wishes to know if an error has occurred. Thus, in order to detect errors during a transfer of data such a test would have to be done after each read or write of the device.

There are certain disadvantages to following each reference to a device with a test for errors and a conditional execution of an error recovery procedure. The insertion of this material into the I/O control program makes it more bulky, and clutters up the written form of the program, making it harder to comprehend its structure. In addition, the explicit and repeated test for errors is expensive. The occurrence of an error is supposed to be the exception rather than the rule, so the ideal mechanism for error detection would involve no cost except when an error occurred.

One solution is to handle error detection the same way that the system handles other errors related to memory references. Examples of this sort of error, which is often called a <u>fault</u>, would be the user attempting to reference a non-existent address, or an address to which he has no access.

Traditionally, what happens if a fault occurs is that an error signal is returned to the processor. The programmer, in order to take advantage of this signal, provides in advance a section of code to be called in case of an error. When the signal arrives, the system will cause this code to be executed. In order to use this technique for dealing with I/O errors, it is only necessary for the device to generate this error signal whenever a synchronous error occurs, i.e., whenever certain bits in the state word are signalled.

This scheme for error detection and recovery is operationally equivalent to the explicit test described above. In the one case, the programmer makes an explicit test at each point an error could occur, and on detection of an error transfers to error recovery code. Using the error signal, he does not make the test, but nonetheless, if an error occurs a transfer will be executed to the error recovery code, and this transfer can occur only at the places in his program where otherwise he would have had to insert an explicit test.

The language PL/I attempts to integrate this technique of error recovery into the syntax of the language by means of the condition mechanism, which the interested reader should study.

The mechanism so far described deals with errors in the restricted case that they are synchronous. The other class of errors, asynchronous, will now be considered. Included with asynchronous errors will be certain randomly occurring events which are not errors, but which must be processed in the same fashion. Examples would be the user pressing the attention key on his terminal or an operator signalling that a tape has been mounted and is ready. Since these signals must be handled much as randomly occurring

errors are, they will be grouped with these errors, and will be called, collectively, asynchronous events.

The error signal mechanism was so far designed as an exact equivalent to the explicit test after each reference. If we attempt to use the error signal to handle the case of the asynchronous event, this equivalence no longer exists. Whereas in the previous case, the signal (and the resulting subroutine call) can occur only at certain explicit points in the program, the asynchronous event could cause the subroutine to be called at any arbitrary point in the program.

This arbitrary interruption is undesirable first, because of the effect it has on the process structure (discussed in the previous sections), and second because the system must provide a fairly complicated piece of software to implement the diverting of the process from its current task to the error subroutine. To avoid these difficulties, a scheme will be proposed which will preserve an orderly structure for the I/O process, and will replace the above mentioned piece of system software with two simple interprocess signals, start process and stop process.

The scheme to be used is the creation of an event process associated with the I/O task. The sole purpose of this process is to detect and respond to asynchronous events. The event process could, quite simply, detect events by looping on a repeated examination of the state word of the device; more efficiently, the device could generate an event signal, distinct from the error signal, which could cause the event process to come into execution. In either case, since the event process puts itself in a known state, looping or waiting for the event signal, it is true here, as it was of the synchronous error, that signals from the device do not arrive

at arbitrary points but at specific locations in the computation, thus arbitrary interruptions due to device signals are eliminated.

The thesis has argued in considerable length the evils of interrupting a process at an arbitrary point in order to execute some other piece of code. Perhaps a specific example will demonstrate the advantages of the separate event process.

As a result of an asynchronous event, it is often desirable to modify the I/O process, perhaps to discontinue the task the process is currently doing and start something else.

To understand the relative complexity of such a modification with and without the event process, consider the dynamic allocation of storage to procedures running in the I/O process. Normally, there will be a stack of activation records, containing the dynamic storage (automatic variables, return information, etc.) for each procedure with a call currently outstanding. If the event handler is a subroutine in the I/O process, when it is called its activation record must be placed on top of the stack. This makes more difficult the task of the handler, for if the handler wants, for example, to abandon the current computation and start a different one as a result of the event, the handler must remove and add activation records to the stack, all the while keeping its own record intact. This means removing and adding items to the middle of a stack, which is complicated. In contrast, if the event handler subroutine is running in a separate event process, it can modify the I/O process without affecting its own execution. To use an old expression, the use of the event process allows the handler to avoid the risk of cutting off the branch it sits on.

Interprocess Signals

The fact that signals from devices do not cause process interruptions does not mean that there is no need for asynchronous signals directed to a process. It just means that these signals do not come from I/O devices, but from other processes. To see the need for such signals, consider the previous example, in which the subroutine responding to the event, running in the event process, wanted to modify the I/O process. Clearly, the I/O process must be stopped before it can be modified. Thus there is the need for an interprocess signal, to be called stop-process, which the event process can send the I/O process before modifying it. Similarly, there must be a start-process signal, for use after the modification.

Is it necessary to have a signal as powerful as stop-process, which actually forces the receiving process to stop? Might the same effect be produced with a passive rather than an active mechanism, a flag which one process would set, and the other would test periodically, stopping if the flag were set? To see the need for an active signal, remember that one event which may occur is the user pressing the attention key on his terminal. The usual meaning of this event is to stop the user's computation, and to place it in a known state. One function of this event is to halt a process which is operating in error. Since there is no guarantee that a process operating erroneously will ever look at a flag, it is necessary that there by a way to force the process to stop. Thus a signal with the power of stop-process is required.

The complexity of the stop-process signal is, however, much less than the signal which is needed if the event process is not present, in which case the signal (which would come directly from the device) must trigger

the following actions by the system. First, it must stop the process. Then it must identify the subroutine which is to be executed in that process. Then it must modify the environment of the process (hopefully in a reversible fashion) so that the subroutine can execute successfully, then it must cause the subroutine to be started. Comparison of these steps with the simple effect of the stop-process signal shows the great simplification in the mechanism the system must provide if the event process scheme is used to avoid asynchronous diversions.

The simplicity of the stop-process signal means less complexity to the system, but it also means more flexibility to the user, because he is then free to use whatever mechanism seems most appropriate for each event. In certain cases a passive device such as a flag may be quite sufficient. Or in certain cases the event process may be able to handle the event without involving the I/O process at all. The flexibility of allowing the user to choose the tools best suited to the task is denied if the event process is not used, for in that case the effect is always the same: the I/O process is diverted to a specified subroutine.

The distinction then, between error recovery with and without the event process is the following. In both cases the I/O process may receive a signal at a random time, but if the event process is used, the signal will come from that process, rather than directly from the device. The advantages of having the signal come from the event process are first, that the signal is much simpler in nature, just stopping the process rather than diverting it, and second that the user has some control over when and if the signal is sent, for the event process executes a program which is provided by the user, so that it can be tailored to the particular needs of the

given event. This ability of the user to control the exact effect of an asynchronous signal is the real advantage of the event process scheme.

Stopping a Process

It was implied in the previous section, that the user, in creating the program to run in the event process, might wish to exercise some control over the time at which that program signalled the I/O process, or any other process, to stop. This control is needed because, while it is very easy to stop a process dead in its tracks using the stop-process signal, it can be very difficult, under certain circumstances, to stop a process in an orderly way such that it can be restarted, and such that other processes, including the event process itself, are not disabled by the original stoppage. This section will explain in some detail the problems of stopping a process in an orderly fashion. In subsequent chapters specific techniques will be introduced by which the I/O process can be stopped.

There are two reasons why it is important to understand this difficulty in bringing a process to an orderly halt. The first is to gain insight into the problems of interprocess co-ordination, and to identify certain other mechanisms which the system must provide to foster this co-ordination. The second reason is to develop the understanding necessary to assure that the I/O process in particular can be stopped in an orderly fashion. That is, the topic of this thesis is the I/O subsystem. It is beyond the scope of the thesis to provide a strategy for stopping any process in the general case, but it is important, as part of the thesis, to solve the specific case of stopping the I/O process in an orderly fashion.

A process which has been stopped dead in its tracks may represent a problem because it may have resources, (devices, data bases, etc.), claimed to itself. If a process holding a resource is to be stopped dead, some other process must find all such committed resources and free them. The first problem is to find them. Unless the system has a uniform mechanism for registering resources, there may be no way to do so. Next, the resource may have been in an inconsistent state at the time of the event, in which case it must be put in a proper condition again. If it is not, some other process, including the event process itself, may attempt to use this resource and discover that it cannot do so. For example, at the point of stoppage the process might be rethreading a linked list. If the event process attempted to use the list, inconsistency threaded pointers might cause the process to loop or to be unable to find some desired object in the list.

In general, the event process will not know how to remove these inconsistencies, for the other process might have been using any arbitrary resource, and the event process cannot know how to restore every resource in the system. It must therefore utilize some procedure associated with the particular resource, which knows how to put that resource in a consistent state. This recovery procedure could be run in the event process, or it could be caused to be run by the other process. In either case there are significant design problems, for example, to certify the reliability of the recovery program. Can the event process trust it? What if it refuses to return? Does the event process give up and leave the resource in an inconsistent state? If so, can it usefully tell anyone? And so on.

Further, there is no reason to think that all resources can be made consistent in the middle of an arbitrarily interrupted modification. In this case the other process cannot be stopped instantly, but must be allowed to run until it has the resource in a consistent state. This privilege of running to a consistent state is, for example, usually allowed system programs called by the user. In this case, since the other process cannot be stopped dead, some mechanism must exist to detect when consistency of all resources exists, and cause the process to stop at that point. This means, in general, that any procedure which requests the right to run to a consistent state must agree to stop. But what if the procedure doesn't stop? For how long should it be allowed to run? What procedures should be allowed to claim the right to run to a consistent state?

Because these questions relate to general issues of resource control and interprocess communication, this thesis cannot propose a solution. It is an important part of this thesis, however, that it be possible to stop the I/O process in particular. In order that the I/O process be stoppable, it is necessary to impose various restrictions on it, the most important of which is that at all relevant times all resources in use can be identified, and can be reclaimed in a consistent state.

Until this point in the thesis, no restrictions have been imposed on the I/O process. While the primary role of the I/O process is to run the I/O control program, there is nothing which would prevent the user from causing it to run any other task he wished. In order that the thesis be able to discuss stopping the I/O process, it will be assumed that the I/O process is restricted to the I/O control function, so that no other unrelated resources need be of concern.

Various other restrictions on the I/O process will be identified throughout the thesis, as goals of timing and efficiency are factored into the system. The general effect of these various restrictions will be to make the I/O process somewhat simplified, compared to the normal process on the system. A by-product of this simplification will be that the I/O process is easier to stop. This will be discussed at several points in the thesis.

A Look Behind and Ahead

To review what has been done in this chapter, an I/O architecture has been constructed which gives the user direct access to his device, which gives him a multiple process structure to organize his I/O task, and which gives him a uniform and coherent technique for error recovery. The important features of this system are:

- The user refers to his device as if it were a segment in his virtual memory.
- The I/O task is implemented in a special process, the I/O process, which is synchronized with the rest of his tasks using some known interprocess communication techniques.
- An event process is provided to detect asynchronous errors, so that no process is ever interrupted by a device at a random point.

The chapter, in addition to describing these features in detail, has discussed the interface which would result between the device and the device selector, and presented an example of a simple I/O control program which might be used in this system.

The chapter has dealt, to a large extent, with issues of processes and interprocess communication. This is because I/O is tied in a very strong way to ideas of synchronization and parallelism. In particular, the idea of an asynchronous error or event, which is very basic to I/O, must be handled in an orderly fashion if the resulting system is in turn to have an orderly structure. The event process is the tool provided to deal with these events, and its generality is felt to be sufficient that the issue of errors and error recovery will be largely ignored in later chapters.

The defect of the I/O system, as developed in this chapter, is that it fails to cope with two problems: first, that devices have real timing constraints, and second, that processors and memory must be used in a somewhat economical fashion. The rest of the thesis will concern itself with factoring these two issues back into the system, without in the process destroying the desirable features which the system now has.

To give the reader a preview of the rest of the thesis, the following is a brief list of the features to be added to the I/O system. There will be one modification to the configuration of the system modules as now described: buffers will in certain cases be inserted between the device and the device selector. The result is pictured in Figure 2-9, which should be compared with Figure 2-1. The buffering will be used to cope with both problems mentioned above, timing and efficiency. In addition, there will be three modifications to the operating system to allow it to interface to the I/O subsystem. The first is a specialized contiguous storage memory allocation scheme, which will work in conjunction with paging to allow memory to be used efficiently. The second modification, to be used in conjunction with the first, is an interface to the virtual memory manager

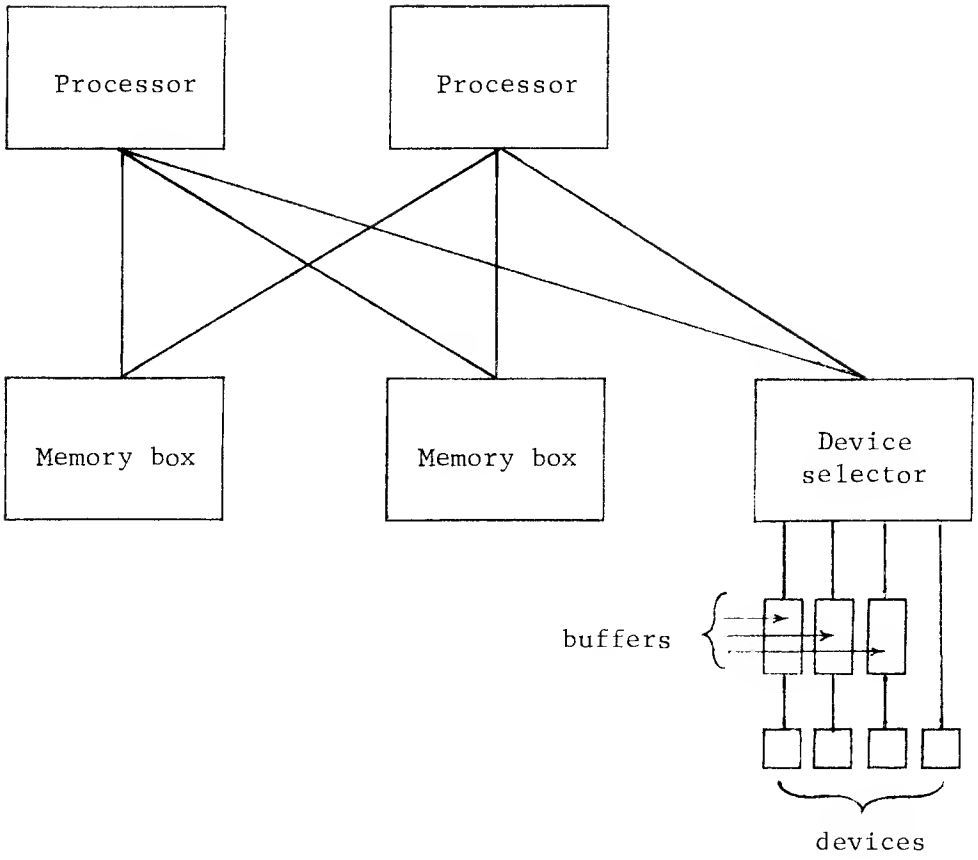

Figure 2-9: Module interconnection with buffers added.

which allows the I/O control program to fix needed parts of virtual memory into real memory in a controlled fashion. The third modification is an interface to the process scheduler which allows a signal from a device to cause a process to be run. The justification and explanation of these features is the subject of the rest of the thesis.

Chapter 3

Interface to Record Oriented Devices

One of the assumptions which was central to the I/O architecture developed in the last chapter was that there were no timing constraints imposed by the devices themselves. In this and the next chapter, this assumption will be removed, and a system will be developed which can deal with devices having real timing constraints.

There are various sorts of timing characteristics which a device might have. Perhaps the most severe, or difficult to interface with, would be a device which generated values at arbitrary times, and which never stopped. The system proposed in this thesis will not interface to devices with arbitrary timing characteristics such as this; rather, an interface will be specified for certain classes of timing characteristics. In particular, it will be necessary to know either the maximum number of items which the device will transfer before stopping, or the maximum rate at which it will transfer them. If indeed the device does not stop, then this thesis will impose an upper limit on the rate of transfer of the device.

In this chapter the class of devices to be considered is that which is often called "record-oriented" devices. A record is just a collection or sequence of a known number of data items which is treated as an entity by the device. By this is meant that once the device has started to transfer the items in a record, it cannot be stopped or excessively delayed without causing an error until all of the record has been transferred. Normally, the items in a record will be transferred at a fixed

and regular or "synchronous" rate, and this rate cannot be modified or delayed without loss of data. The allowable delay is usually specified in terms of the rate. For example, the transfer of each item might have to be completed before the next is initiated or the item from the first transfer will be lost. Devices with these general characteristics will be called "record-oriented". It will be shown that such devices are relatively easy to insert into the I/O system of the last chapter, so this class will be investigated first.

The purpose of this chapter is to devise a scheme which will avoid the disruptive effect of delays to the I/O process. There are two general solutions possible. The first, to be called the external solution, is to insert some adapter between the device and the port on the device selector so that delays in the I/O processing can be tolerated. The other approach, to be called the internal solution, is to modify the system so that no unacceptable delays can occur. There is a third solution, which is to start the data transfer over from the beginning of the record if it is disrupted by a delay. If there is some assurance that the same delay will not reoccur on each attempt to transfer the record, this solution may work, but it is not general, for it only works for certain devices. This solution is discussed more fully in the next chapters.

The most obvious form that the external solution could take would be a buffer, to hold the items which need to be transferred while the system pauses for any reason. At first glance, the external buffer might seem the simpler of the two solutions, for in the internal approach one must show that all causes of delay have been found and eliminated. Closer inspection, however, will reveal that buffers have certain disad-

vantages, the two most important of which are that some facility is required to synchronize the inflow and outflow from the ends of the buffer, and that some facility is required to recover the data which may be left in the buffer after an error has halted I/O.

The advantage of the internal approach is that it yields a much simpler structure; the buffers and their associated problems do not exist. For this reason the internal approach will be pursued first. In the next chapter the external approach will be explored, and these problems with buffers will be analyzed in greater detail.

In this I/O system, the most obvious delay which might disrupt the transfer of a record would be caused by the virtual memory manager pausing to fetch a page involved in the I/O operation. That is, the I/O control program could cause a page fault, or page exception. If the time to fetch a page is greater than the maximum allowable delay of the device, then data will be lost. This problem did not exist in the earlier system, in which devices had no timing restrictions. Essentially, this chapter will present a solution to the page exception problem. There are, however, other possible delays, and all must be dealt with.

As a beginning to the elimination of all interruptions and delays which an I/O procedure might encounter, let us construct a list of various sorts of delays which are observable in systems today. Delays to the I/O control program might be sorted into four classes.

1) Processor multiplexing (time-sharing, multiprogramming)
2) Interrupt handling (I/O, time, etc.)
3) Error handling and recovery
4) Virtual environment modification (dynamic changes)

The first three of these are easy to eliminate. By the assumption that a processor can be dedicated to a process doing I/O, the first case no longer exists. Similarly, the second case can be eliminated. In this system no I/O interrupts exist. The other interrups, such as an interrupt from a timer, can be dealt with in the same fashion as I/O device signals were, by directing them to a process which is waiting for them. This disposes of the second category. As for the third category, it is obvious that the progress of an I/O program may be delayed or disrupted if the user program contains an error, but such delays are not a defect of this scheme; user errors can be expected to disrupt the user's computation. The only responsibility that the I/O system must take is to assure that the results of a delay due to an error do not obscure the nature of the error itself. This leaves the fourth category: dynamic modification of the virtual environment.

What is meant by a dynamic modification is any modification to the environment made "on the fly", at the point in time at which that modification is needed in order for the user's program to continue. An example would be a page exception, in which the page is fetched into real memory at the moment the user needs it. Clearly, since dynamic modifications are made "on the fly", the user's process will experience some delay while they are performed; it is this delay which must be eliminated. (In specific cases, where the delay occurs at a known point and takes a known time, the user could allow for it, but in general delays do not occur in this predictable fashion.)

The sort of delays which can result from changes to the virtual environment depends, of course, on exactly which aspects of the environ-

ment can change dynamically. The various changes can be divided, however, into two categories. The first category is the change in the binding, or relationship, between two virtual namespaces. For example, in the Multics system, the first time a procedure references a segment by name, a binding is created between the segment name (one virtual memory) and a segment number (another virtual namespace). This binding is called linking in most systems; in Multics it is done dynamically. The other category of dynamic changes are those which bind a virtual to a real name. The most obvious example, of course, is the association of a virtual address with a real memory address, which is the result of a page exception.

The I/O system can easily eliminate delays of the first sort, for the cost of making such bindings in advance is not the tying up of real resources, but just the expense of identifying and making all the bindings. In the case of dynamic linking, for example, it is quite reasonable to provide a static linker, and require the user to employ it on his I/O program.

The difficulty comes with bindings of the second kind. When a binding from a virtual name to a real resource is created, it implies that the real resource so bound is committed for the duration of the binding. This committment is costly, so that such bindings are only made and kept for as short a time as possible. A page of a segment, for example, is brought into memory (bound to real addresses) only as needed. This implies that if no pauses are allowed during I/O, so that these bindings must be made in advance, a real cost will be incurred for the real resources. The purpose of this chapter is to understand how to con-

trol these costs.

For convenience, let us call a binding which has been completed and which will be maintained in that state a frozen binding, and an environment with all appropriate bindings frozen, so that it might support I/O, a frozen environment. There are two questions to be considered concerning the freezing of a binding, or the freezing of a page in particular: what is the impact on the system, and what is the impact on the user? The next sections will consider these points.

The Effect of Frozen Pages on the System

From the system's point of view, what restrictions must be placed on the user's ability to ask that various of his pages be frozen in memory. As this section will show, there are three criteria which the system must guarantee in order to assure successful operation. They are as follows.

First, the real cost, as paid by the user, of fixing a page in memory must not be so high that he cannot afford to do I/O. This particular issue will for the present be ignored, because questions of cost and efficiency have been postponed to a later chapter. In particular, Chapter 7 will reconsider the material in this chaper, and will augment the scheme to be described here in such a way that the cost is made acceptable. For this chapter, it will be sufficient to assume that the user does pay whatever real costs are associated with his frozen environment.

Second, an issue distinct from, but related to the first, the user must not be allowed, by means of freezing pages in memory, to claim more

than his share of the machine. It is clear that a user could, by freezing large numbers of his pages in memory in an uncontrolled fashion, use up so much memory that other users of the system had insufficient memory left to run well. The user must not be allowed to optimize his computation at the expense of overall system performance, even if he is willing to pay for the resources he freezes in doing so. Thus the issue of fair share is distinct from the issue of cost.

Third, the freezing of pages into memory must not interfere with the ability of the virtual memory manager to perform other necessary tasks. From time to time, the system needs to undo virtual-real bindings. For example, if it is desired to reconfigure the system by removing a memory box while the system is running, then any pages bound to memory in that box must be unbound and moved. Unless one assumes that reconfiguration is so fast that the delay it cause is negligible, which is not a realistic assumption, then the frozen binding is an effective obstacle to tasks such as reconfiguration.

Both of these latter problems can be solved by imposing the following simple constraint on the freezing of pages: each request for frozen pages must be accompanied by a specified maximum time which the freeze must continue in effect. The user making the request will determine the appropriate time, and the system will hold the user to this limit. In order to understand the implications of this time limit, observe how it solves the reconfiguration problem. If the system can guarantee that a frozen binding will come unfrozen before some particular time, and the delay until that time is short enough, then the process which is performing the reconfiguration can, on discovering that it can-

not move a page because the page is frozen, request that it be notified when the page becomes free, and then suspend its execution, knowing that by virtue of the time bound the system can guarantee that the page will actually be freed. (In order that the reconfiguration process be able to tolerate the delay, the time limit on frozen bindings must be fairly short, perhaps a few seconds at most.) Thus reconfiguration can succeed even with frozen pages.

The time limit on requests similarly provides a means to solve the fair-share problem. In order to see why the time limit is important, note that the share of memory represented by two requests, one for a large number of pages for a short time, and one for a small number of pages for a long time, is in some fashion equivalent. That is, resource consumption is a space-time product. This being so, and given that the time limit on frozen environments means that both the time and the space commitment represented by a given request can be determined at the time the request is made, the time limit allows the memory manager to restrict each user to his fair share of the system resources by taking each request for a frozen environment and not granting it until sufficient time has passed so that in granting the request the user gets no more than his fair share. The user with a large request would thus discover that his request was granted only after a long delay.

The time bound is very important in making this scheme work. The resource scheduler would not be able to assess the resource consumption represented by a request if that request was not associated with a maximum time. Without the time limit, a resource once frozen might never be released. Thus the time limit is crucial in regulating the resources

consumed by I/O.

It has thus been shown that the system can function properly, even allowing for frozen pages, if a time limit is imposed on the duration of the freeze. We must now turn from the system to the user, and consider what effect this frozen environment scheme with time limit has on the user's ability to perform I/O.

The Effect of Frozen Bindings on the User

There are two questions about the effect of the frozen environment scheme on the user. First, does the requirement that the user predict in advance the maximum time that his I/O task will take restrict him in such a way that he is prevented from doing useful work? The answer is that the time limit is completely compatible with certain kinds of devices, in particular record-oriented devices, because such devices transfer a known number of items at a known rate, so that the transfer requires a predictable time. For other kinds of devices, such as typewriters, which, at least in input mode, are essentially unpredictable, the time limit scheme is not applicable. In Chapter 7, the scheme will be modified to work for typewriters and other such devices as well.

The second question concerning the user and the frozen environment is what modification in the structure of the I/O system developed in the last chapter is implied by the use of the frozen environment? Whenever a user is about to perform I/O, he must now precede the I/O with a request to freeze his environment, and follow it with a request to unfreeze. What complication will this represent to the user? The addition of two subroutine calls to the I/O control program is not a major com-

plication. The real complexity is determining which parts of the environment need to be frozen, and describing these parts to the system. In general, the following sorts of areas will be needed:

- the procedures doing I/O
- storage for variables of the procedure
- storage for I/O data being transmitted

What aids can be constructed to help the user identify the parts of his environment to be frozen? If the procedures are written in some high-level language, the user will not know which areas of memory contain the portions of his program which are to be executed during the freeze. It would be possible, however, to devise a procedure which would trace the user's program and generate a list of those areas which must be frozen. All the procedure need do is start at the call requesting the freeze and follow each branch until it finds a request to unfreeze. This procedure can also generate a list of those variables whose storage must be available. The principal difficulty comes with the storage for the I/O data itself, for only a certain part of this storage need be frozen on any particular transaction. The storage might, for example, be represented as an array, a certain area of which will be referenced. The array may be very large, which would make it unacceptable to freeze the whole array in memory. It would be very difficult for some run-time procedure provided by the system to deduce from the program and the current value of variables just what portions of the array will be referenced. For areas such as this whose boundaries change with each instance of the freeze, it is probably necessary (and desirable, from the viewpoint of efficiency) for the user to construct explicit expressions describing the particular

areas of the variable storage which should be frozen on each I/O transaction.

In summary, then, the I/O system of Chapter 2 is modified as follows. Before starting an I/O transaction, the I/O program must call the memory manager, presenting it a list of those resources which must be frozen for the transaction. Inside this call the I/O process will pause until the manager decides to honor the request and freezes the resources. When the resources have all been frozen, the manager will set a timer with the limit which was supplied as part of the request, and return to the user. When the user has finished the I/O, it must call another entry in the manager, which will free all the frozen resources. Should the timer run out before this call has been made, the manager will assume that the user program is in error, and will stop the I/O process, free the resources, and notify the process overseeing the computation.

The most obvious limitation implied for the user by these controls is that he must agree in advance of each transaction to a maximum time limit. This means that (until Chapter 7) only certain kinds of devices are connectable to the system. There are certain other drawbacks. For example, since all the areas to be referenced by an I/O transaction must be specified before that transaction begins, any data which contains the address into which the rest is to be read cannot be read in one transaction (except into some intermediate area).

Other Bindings

The freezing of pages into memory has been stressed here because these bindings are expensive to create and maintain, but the other

bindings must not be forgotten. One must examine a particular system to find all the relevant bindings, but some general sorts of things to be expected can be listed.

- All segments to be used must be made part of the virtual address space.
- All symbolic references must be linked to the correct segment.
- Any process to be signalled must be identified.

In fact, since the process cannot add segments to its address space, or reference new portions of known segments, its restrictions resemble those felt by programmers before the invention of dynamic creation of bindings, when they were forced to define in advance the requirements of their computation. But of course this is exactly what is to be expected of a frozen environment.

Stopping the I/O Process

In the second chapter it was stated that the various restrictions which would be placed on the I/O process would allow us to state how another process (e.g., the event process) could stop the I/O process in an orderly fashion. In what ways have the controls of this chapter taken us toward that goal? First, note that a running process is in one of two states, frozen or unfrozen, and unless it is frozen it cannot be performing I/O. If it is not performing I/O, there is no reason why it cannot be stopped instantly. If, at the time it is to be stopped it is doing I/O, then all the physical resources which it is holding are named in a list, the list describing the frozen environment, which is known to the system as well as to the I/O process. Thus if the I/O process is to

be stopped instantly, it is only necessary to stop the I/O process itself, make sure the device is stopped, and then tell the system to free the listed resources.

Alternatively, if appropriate, the I/O process can be allowed to run until the I/O is completed. There are essentially two problems with allowing the I/O process to continue in this fashion: how to tell when the I/O is completed, so that the I/O process can be stopped, and how to determine that the I/O process is operating in error and is not ever going to stop. The frozen environment scheme with time limit provides the ability to solve both these problems. First, when the I/O process finishes the I/O it must call the system to unfreeze its resources. At this point the system can, if appropriate, stop the I/O process and notify the event process. In this fashion the event process can get control of the I/O process at the exact point the I/O is completed. Second, the event process should conclude that the I/O process is behaving in error just when it overruns its time limit. In fact, the obvious way for the system to deal in general with the failure of the I/O process to unfreeze its environment within the time limit is to stop the I/O process and notify the event process.

Other Forms of the Time Bound

It should be clear that if the system is to keep any control over resource usage, there must be some sort of time limit on the freezing of resources. It may not be obvious that the rather simple form of the bound used here is the most appropriate. This section will explore some of the aternatives.

We might first inquire if instead of an absolute time limit, some more flexible bound could be used. The bound could be in the form of a probability distribution of the time which will be used. This thesis has not used such an idea because the advantages, especially to a record-oriented device, are nil, and the complexity of the scheme is significant. The only way in which the system can set a timer using a bound in the form of a distribution is to assume that the I/O process will use the maximum time the distribution will allow. The system can only use the distribution as a scheduling tool in estimating resource usage if it has confirmed that the I/O process is indeed conforming to the distribution it supplied. The measurements necessary to confirm this would be costly. Finally, for a record-oriented device, where known record length implies fixed time bound, there is no obvious reason why a distribution is an appropriate description of the time the I/O process will use. So this thesis will consider only fixed time bounds.

There is another variation on this scheme, however, which seems more useful. This is to change the interpretation of the time bound, so that it describes the maximum time within which the process will unfreeze, not counting from the time the freeze goes into effect, but rather from the time when the resource manager asks the I/O process to unfreeze its resources. This technique is especially appropriate if the fair-share problem is not important, so that the only need for the time limit is to allow for reconfiguration and other rare occurrences, in which case it would be more efficient to let the I/O run until a particular need to unfreeze actually arose, rather than making the I/O repeatedly stop to unfreeze and then refreeze its environment.

The disadvantage of this variation is that the signal from the resource manager will reach the I/O process at an unpredictable time, which is undesirable. To avoid the necessity of this signal, which is essentially an asynchronous interrupt, while still providing a means by which the I/O process can run until such time as the resource manager wants it to stop, a new entry point to the resource manager can be provided which the I/O process may call to extend its time limit whenever the current time allotment is about to run out. The resource manager can then stop the I/O as necessary by refusing to extend the bound. This approach does imply the extra cost of these calls, but it avoids the design of the communication path from the resource manager to the I/O process. This thesis will presume this new entry point if such a variation is appropriate.

Summary

This chapter has proposed that in order to interface to a particular class of timing dependent devices, namely record-oriented devices, a modification to the virtual memory manager be made so that pages of the user's virtual memory can be fixed, or frozen, into real memory during I/O. In order to regulate this fixing of pages into memory in a manner suitable for both system and user, the chapter restricts the freezing of pages by requiring that the user specify, as part of each request to freeze pages, the maximum duration of the freeze. This restriction, although very simple, is powerful enough to solve two problems associated with frozen pages: assuring that the user can get no more than his fair share of the machine, and assuring that the system can move frozen pages as needed. The time limit has the additional benefit that it constrains the I/O pro-

cess in such a way that it is possible for the event process to stop the I/O process in an orderly fashion if necessary.

The freezing of pages is actually a particular example of the need to eliminate all delays which can occur while doing I/O. The chapter discusses the various sorts of delays, but concentrates on paging, the most difficult delay with which to cope.

In fact there were three controls placed on the user's ability to freeze pages in memory. Not only must the user supply a time limit, but he must pay for real resources so committed, and he is scheduled only so often as gives him no more than his fair share of the machine. Of these three controls, however, the time limit is the most important, for without it the other controls would not function properly.

One of the goals of the thesis was to seek solutions which were universal in scope, that is, to find solutions which applied to a wide variety of devices, rather than to one particular device. While the freezing of pages cannot be called a universal solution, for it applies only to record-oriented devices, it clearly applies to a significant class of devices, and in Chapter 7 it will be extended to include a wide variety of devices. The frozen environment is thus an example of this desirable sort of solution.

Chapter 4

Buffered Interfaces

In an attempt to interface devices with timing constraints, two classes of mechanisms, internal and external, were proposed. The last chapter discussed the internal approach; this chapter will discuss the external approach. The internal approach attempted to interface devices with timing constraints by systematically eliminating all possible causes which could delay the I/O control program during those periods when the device is operating. In contrast, the external approach interposes a buffer between the device and the port on the device selctor, which allows the device to tolerate any delays of the I/O control program by providing storage for the items which must flow during the delay.

Buffering in connection with I/O is far from a novel idea. The intent of this chapter is not to present new ideas about buffering, but to demonstate the real cost and complexity of using buffers as part of this or any other I/O scheme. In brief, the chapter will show that buffers can be used in the context of this I/O architecture and that the I/O device can still be represented as a number of memory words but will also show that the I/O control program must take explicit account of the buffer, especially in error recovery, and that the complexity added to the I/O control program by use of buffers may, depending on the particular device connected, be considerable.

In order to show the issues involved in buffering, this chapter will develop a particular buffering scheme, the role of which is to serve as an existence proof that such a buffer can actually be build. The scheme

is by no means unique, or even best, and the chapter will point out alternatives which might be appropriate.

Buffers are needed to deal with devices which are not record-oriented, but which have other sorts of timing characteristics. In order to motivate this material, these other characteristics must be introduced and understood. Thus this chapter will begin with a discussion of various kinds of devices.

Most devices which are connected to computer systems are record-oriented, including disks, drums, card readers, and card punches. Devices such as printers and tapes have variable record sizes, but the maximum size is always known before the device is started. The largest class of devices which are not organized around records are often called communication devices; these include communication lines to distant devices or machines, and devices such as terminals and other devices for human interaction (graphic displays, light pens, input tablets, etc.). Since an interactive terminal is a very common device, crucial to time-sharing systems of today, and since the timing characteristics of such terminals are fairly typical of these sorts of devices, interactive terminals will be considered in more detail. On input the terminal generates an item whenever the typist presses a key. It is considered bad human engineering to prevent the user from typing as he wants to, so terminals are normally prepared to accept a character at any time. Thus, in contrast to a record-oriented device, which delivers a known number of items at a fixed rate, a terminal delivers as many items as the user chooses to type at whatever rate the user chooses to type them. If the user were entering a large quantity of text into the system, the terminal might

continue accepting input characters uninterrupted for hours. On output, the device does not demand items at any fixed rate, so it could be used as a record-oriented device by delivering output in blocks of fixed size; however, output which comes in bursts like this is apt to prove almost as annoying to the user as having his keyboard locked when he wishes to type. Another observation about terminals and similar devices is that the transmission rates are (at least for input) usually much lower than those of the typical record-oriented device, so that the resources needed to run them at full capacity are not as great. Issues of resource consumption are being postponed until a later chapter, but, looking ahead, one of the requirements of any scheme for operating these sorts of devices must be that the scheme not require excessive resources.

It is not true, strictly, that devices such as terminals do not stop. It is almost always true, except in systems dedicated to the I/O task and specially designed, that the system allocates resources to the device using probabilistic rules that attempt to serve the device successfully "almost" all of the time. But it is generally accepted that occasionally the system can and will fail to provide resources when the device needs them. In this case the device will be forced to stop (for example the keyboard will be locked). Thus while it is considered bad design to lock the user's keyboard continually, rare interruptions are acceptable. This tolerance of occasional interruptions can be exploited in the system design.

How might a device such as a terminal be connected to the system? The frozen environment, with its limit on the length of the transaction, is an inappropriate interface for a device with unpredictable timing

characteristics. The interface could be used under one condition: if the system did not use the time bound as a scheduling tool, but only to allow reconfiguration and similar tasks. Since these tasks are rather rare, it might be reasonable to accept the pauses for them as the occasional tolerable disruptions. For the purpose of this thesis, such an assumption is not very general, so another solution will be sought.

The solution to be discussed in this chapter is, as was mentioned above, the insertion of a buffer between the device and the system, as illustrated in Figure 4-1. The buffer used in this fashion provides an area in which data in transit can 'pile up' to cope with delays induced by page exceptions or other causes.

This buffer could appear in various forms. It could, as pictured, be a separate module which is physically inserted in the line between device and device selector. Alternatively, the buffer could be included within the device. For example, many interactive terminals being designed today include a mini-computer. Such a computer could be programmed to use its memory as a buffer. In this chapter the buffer will be viewed as a physically separate module, because this reveals most clearly the issues involved with buffering.

The buffer scheme to be proposed in this chapter must be evaluated in the light of two considerations. First, Chapter 2 described a specific device interface, with command and data lines controlled by sets of ready and acknowledge lines. To what extent can a buffer scheme be proposed which can work with this interface as described? This chapter will show that certain new lines are required. Second, to what extent must the user change the way he views his device if that device is connected by

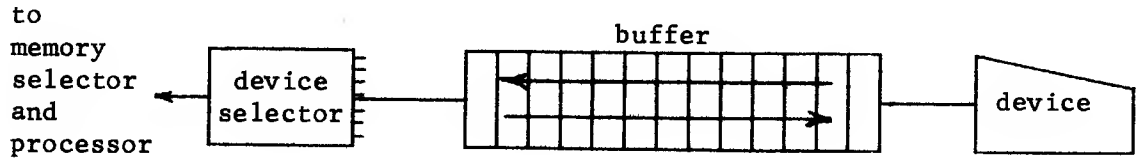

Figure 4-1: Buffer inserted between device and device selector.

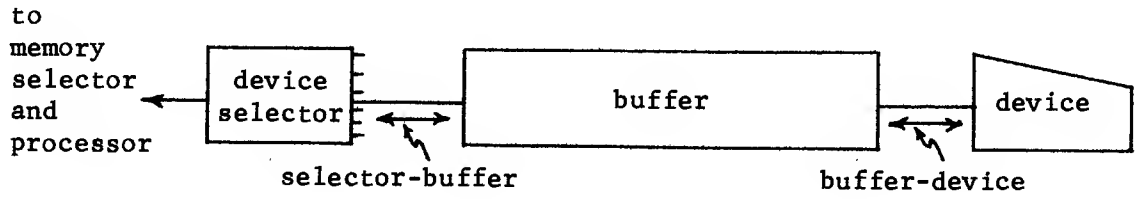

Figure 4-2: The two stages of data flow in a buffered device.

means of a buffer? For example, what changes must be made to the program of Figure 2-4 if the device is buffered? The chapter will show that the representation of the device as a portion of the address space is still an appropriate and viable interface, but that the I/O control program must take explicit account of the existence of buffers.

A Model of I/O Buffering as Several Parallel Algorithms

The introduction of the buffer has broken the flow of data into two stages, between selector and buffer and between buffer and device, as pictured in Figure 4-2. At each of these stages, the flow of data is under control of a particular algorithm, to be called the data flow algorithm. The selector-buffer data flow algorithm is the I/O control program running on the processor; the buffer-device data flow algorithm is provided by the internal structure of the buffer. These two algorithms must work in parallel to move the data the whole distance between the selector and the device.

The picture could be extended as in Figure 4-3 so that there were several buffers with associated data flow algorithms. In this general case, all the algorithms A_1 through A_{n+1} must be coordinated to produce correct effects. This sequence of data flow algorithms A_1 through A_{n+1} could be compared with a bucket brigade in that the successful operation of the overall system depends on the co-operation and co-ordination of the sequence of semi-autonomous operations.

Why would a sequence of buffers as in Figure 4-3 be a preferable model compared to the single buffer of Figure 4-2? Imagine for a moment that we have successfully inserted a single buffer between device and

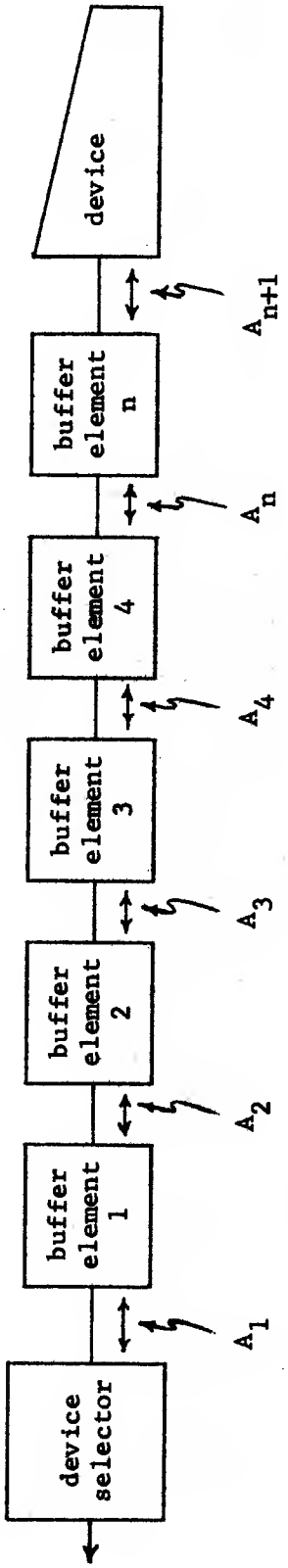

Figure 4-3: Several buffer stages and associated data flow algorithms between device and selector.

selector. If the device continues to operate properly, then it must be true that the buffer behaves exactly like a device as far as the processor can tell, and behaves exactly like the processor as far as the device can tell. This in turn means that if one of the connections to the buffer was severed, say the one between the buffer and the device, and a second buffer inserted there, neither buffer could tell that the other buffer existed, because each would, to the other, be indistinguishable from the device or selector which the buffer replaced. Thus, one need only design a simple buffer element which can hold just one item, for by combining these elements one can create a buffer of any size needed. This approach is obviously much simpler than designing the whole buffer as a unit. Thus, this thesis will consider the buffer to be of the form pictured in Figure 4-3.

It is usually impractical, as well as undesirable, to have the user provide all of the data flow algorithms. Normally devices and buffers are simple devices, which are not programmable. They come from the factory with the algorithm 'built-in'. The subject of this chapter is thus what should these built-in algorithms be, and what co-ordination between them is necessary, so that the system works properly. In particular, the data flow algorithms must be structured such that they solve two specific problems. First, the inflow and outflow of items from the sequence of buffers must be co-ordinated in such a way that the buffer is, as appropriate, kept full or empty. Second, if because of an error the data flow is halted with items in the buffer, the various algorithms must co-operate to empty the buffer and put it in a known state. This chapter will propose a buffer algorithm which solves these two problems.

Synchronization of Buffer Algorithms

In synchronization the inflow and outflow from the buffer, the following considerations must be taken into account. When I/O starts, the buffer will normally be empty. Similarly, when I/O comes to a halt the buffer will normally be empty. The empty state is normal for reading from a device, so that room exists for holding values which the device generates. In writing the buffer should normally be full, so that values are available for the device if the process stops. This means that when writing, the I/O programs, as part of starting up, must fill the buffer, and then attempt to maintain it in that state.

The following is a specific buffer algorithm which will satisfy these constraints. It is based on the particular interface between the device and the selector discussed in Chapter 2, which consisted of command lines running from selector to device, and data lines which could carry data in either direction. Each I/O operation was composed of two parts: first, the command, going from selector to device, and second, the data itself, going in either direction as specified by the command.

Consider reading data from device to processor. In this mode the buffer should be empty, to provide space to hold items generated by the device while the processor pauses. The processor will commence the read operation by issuing a read instruction, which will cause the selector to send a read command to the first buffer. Obviously the buffer must transfer the command to the next buffer, so that it may eventually reach the device. Thus part of the buffer's data flow algorithm must be:

> When empty buffer receives read data command from selector, acknowledge it, and pass it on to the device.

The next step will be the data being read, which will flow back from the device to the processor. Thus the other half of the algorithm is:

> After handling the read command, wait for device to send data back. When data arrives, acknowledge it, and pass it on to the selector.

The next step is to make the buffer work properly in the case that the processor lags behind. In such a case, the device must be able to force an item into the buffer, even if the read data command has not been issued by the selector.

The technique to be used is to add one line which runs from device to buffer, to be called the 'read operation required' (ror) line. If at the time the device needs to transfer another item, it has not received a read data command, it will signal over the ror line to the buffer. The buffer's response will be to create and send to the device a read data command, so that the device can then transfer its data item. The buffer, which must get rid of this item in order to accept another, will force the item into the next buffer by using its ror line. Thus the item will be shifted eventually to the buffer adjacent to the device selector; when this buffer signals over its ror line, the processor will not respond, so the data item will sit until the processor is ready and sends a read data command.

In order to implement this, the buffer algorithm must be augmented as follows:

> When empty buffer receives ror signal from device, fabricate a read data command and send it to device. Send ror to selector. When data arrives from device and read data command arrives from selector, pass data to selector.

For those who are interested in the details of this algorithm, it is

presented in Appendix A as a finite state machine. The algorithm is not simple; to allow for the various possible sequences of events, nine states are needed.

There is another solution, appealing except for a fatal flaw, which does away with the necessity of the ror line by having the buffer, once the sequence of reads has started, always generate a read data command as soon as the previous read is complete. Why does this not work? Because the buffer does not know when to stop. The sequence of reads should come to a halt when the device reaches a logical stopping point, which might be a record boundary or the end of a message. If the buffer were to generate a read data command after this stopping point is reached, the device will continue on to the next record, which should not happen until the processor requests it. Thus the device, which can identify the logical stopping points, must have control over the flow of read data commands, which in this scheme is done by means of the ror line.

This is half of the algorithm. What about writing? Writing requires that the buffer be kept full, so that items will exist for the device if the processor pauses; thus the algorithm here will differ from the read algorithm in that it will try to give the processor a head start over the device in order to fill the buffer up. One strategy which would do this is as follows: when the processor writes an item into the first buffer, that buffer does not immediately transfer the item to the next buffer as in reading, but rather holds it until another item is presented to it. The next item forces the first into the next buffer, and so on, until the sequence of buffers is full, at which point the next item will force the first item out of the buffer and to the device. Once the

device is started, some mechanism is then required to allow it to pull additional values out of the buffer, even if the buffer is not being kept full from the other end. The mechanism required would be a 'write operation required' (wor) line, similar in nature to the ror line, running from device to buffer. When the device signals over the wor line, the buffer will transfer the next item, or raise its wor line to fetch the next item down. In this way the device, once started, can empty the buffer.

There is one difficulty with this strategy for writing. If the length of the message is shorter than the number of buffer elements, the message will never get written, because the algorithm relies on filling up the buffer in order to start the device. One form of solution to this problem would be to have the processor start the device explicitly, but this is undesirable because it would change the model which the processor had of the device. Even with buffers, the processor has been able to reference the device using a sequence of memory references. It would be a shame now to add the necessity of additional commands. Another algorithm will avoid this problem. Design the buffer so that it always passes an item along immediately. This will mean that the device starts as soon as the processor does, so the buffer does not assist in keeping itself full. The buffer can be filled without this assistance if the processor can write items faster than the device can accept them, and each I/O transaction starts with a long enough uninterrupted sequence to fill the buffer. Since the necessary length of such a sequence can be figured, the frozen environment techniques of the last chapter can be used to guarantee such a sequence. The statement of this latter al-

gorithm is rather simple:

> Whenever the buffer contains a data item from the selector, send a write data command to the device, and on acknowledgement, send data to the device. When a write data command arrives from the selector, acknowledge it. Wait until buffer is empty. Then wait for data from selector. Pick up data and acknowledge it.

The algorithm involving the wor line is somewhat more complicated, although the general scheme is clear enough. The interested reader is again referred to Appendix A, where both write algorithms are displayed in detail. It will be assumed for the rest of this thesis that the simpler scheme not using the wor line is preferred.

This completes the specification of an algorithm sufficient to coordinate the various buffers under normal operation. The resultant algorithm is more than a little complex. The resultant algorithm is also by no means unique. There are changes which could be made. Moreover, this particular buffer structure, a sequence of identical buffer elements, is not the only useful structure. The buffer could be a small computer or a single piece of LSI, which implemented the algorithm as a program, or it could be built as an integral part of the device.

One other issue related to synchronization has to do with reading and writing the state word of the device. The issue is the following. When the processor reads or writes the state work, is the action supposed to occur immediately, or should it be delayed in the buffer? This problem is most severe when writing, because the processor can, by virtue of the full buffer, get ahead of the device. In this case if the writing of a state word had immediate effect, the writing of the items currently in the buffer might be severly affected. On the other hand,

if the effect is delayed until the buffer is cleared, then it would be impossible ever to stop the device immediately, which is a bad result. Thus it becomes necessary to distinguish the cases of immediate effect and buffered delayed effect, and this distinction means that the I/O process cannot completely ignore the existence of buffers. One solution to this problem is that if the state word should not be modified until the buffer is empty, then the process should wait until it can determine that the buffer is empty and the device stopped before proceeding. The process thus must be able to detect this condition by reading the state word.

With these various observations we will conclude our discussion of synchronization of buffers and pass to the other structural question raised earlier, the recovery from errors.

Error Recovery with Buffers

The other problem with buffers was that buffers complicate the recovery from errors and abnormal halts, for an abnormal halt may leave data in the buffer which must be dealt with. This situation is confused by the several different cases which can occur. Because of an error the device may stop, or the processor may stop, or for some reason another process may want to halt the I/O in the middle of the transaction. A good example of this latter case would be the event process when it responds to the user's attention key. The task of this process would be to stop the current computation, including any I/O in progress. In order to simplify this discussion, we will first consider this last case, and see in what fashion another process could stop an I/O trans-

action involving a buffer.

The most simple technique for stopping the I/O is to halt both the device and the processor and just discard anything in the buffer. This technique has been used because of its great simplicity, but in a properly designed system such loss of data should be avoided when possible. In response to the signal from the attention key the current computation is certainly to be stopped, but it is very desirable that this stop not be a destruction but a suspension of the process, so that the computation might be restartable. The ability to suspend, modify, and restart a computation is present in the Multics system, and has proven very useful. If data is lost by stopping the process, the process may not be restartable. Thus, a goal is that stopping the I/O process not cause loss of data if possible.

Given this goal, can the simple strategy of discarding the buffer contents ever be used? It can, under one condition: that the sender and receiver of data can agree on some previous point in the I/O transaction, some check-point at which to begin again. Such a check-point often exists. For example, if the device connected through the buffer were record-oriented, the transfer could be restarted at the beginning of the current record. This usually works for disks and tapes, is awkward for card equipment, and unacceptable for printers, because the same material would be printed twice. For a printer, however, it might be acceptable to start over at the previous page boundary, or at the beginning of the file. Whatever the boundary, this strategy is facilitated if the I/O program always allows the buffer to empty before moving from one record to the next. As long as this rule is obeyed, there is never

any question of the record in which the error occurred. Otherwise, it may be necessary to back up further than the current record.

How could the I/O control program discard the contents of an array of buffer elements? An obvious and effective mechanism is to set aside several bits in the state word which are to have meaning to the buffer rather than the device. These bits might be more properly made a separate state word, an interface state word, rather than a part of the device state. These bits would be mapped into the additional lines which are present at the buffer interface. The example so far is ror (and perhaps wor), which runs from buffer to processor. In order to implement this buffer discard function, another line is needed, running from processor to buffer. Setting this line would cause the buffer to reset itself and then pass the signal on. Thus another line is added to the device interface for buffer error recovery. Figure 4-4 shows the device interface with the various lines which have been added in this chapter.

If no check-point can be found, then in some fashion it will be necessary to restart from the point of stopping. One obvious way to stop the I/O so that it can be restarted from the point of stopping is to halt the transmitter of the data but let the receiver continue to run until it has emptied the buffer. This is a variation of the pattern which would occur during normal termination of an I/O operation.

Under what circumstances is this solution unacceptable? It may be that the receiver is what really needs to be stopped. For example, if the current computation is printing large quantities of output on the user's terminal, and the user decides that he does not want this output and presses the attention key to stop it, he does not want his terminal

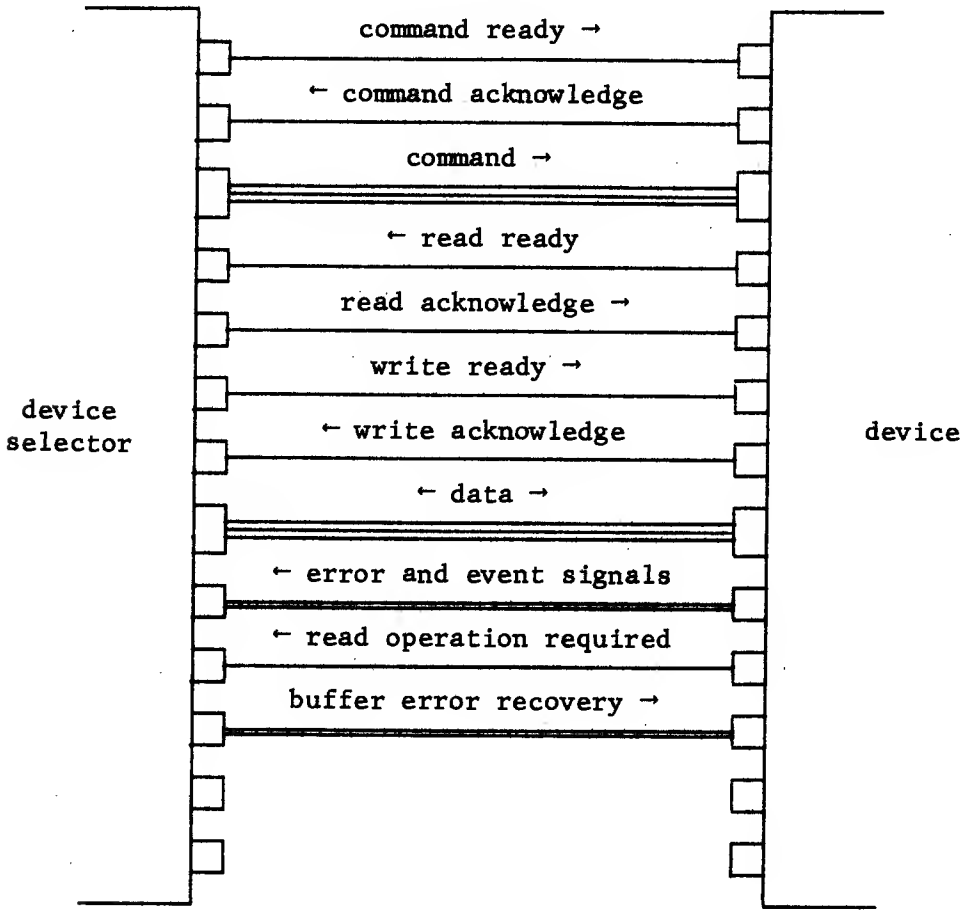

Figure 4-4: Device interface of Figure 2-5 with read operation required and buffer error recovery lines added.

to continue while a potentially large buffer is emptied. He wants it to stop when he presses the button. Allowing it to continue causes frustration, but is also bad interface engineering because the terminal is the only link between the user and the computer. If the computer fails to respond to the attention key immediately, the user has no way of confirming that the attention signal was noticed.

It is normally not necessary to stop the process as quickly as the device, so if the process rather than the device is the receiver, this technique may work well. In terms of the implementation, the only difficulty is that the I/O control program must determine whether there are additional items in the buffer, in order to know whether to issue another read. If the ror line can be tested as a bit of the state work, however, the presence of a signal there indicates exactly that another item remains. Thus there is no problem.

Another reason why the technique of letting the receiver continue to run will fail is that the cause of the stoppage may be the receiver itself, which has halted when it discovered an error. If the error was a bad data item, for example, the item must be replaced with a good one before the receiver can be restarted. Thus the receiver cannot be used to empty the buffer.

The next possible technique for stopping is that the transmitter should take care of the data in the buffer. One simple implementation of this would stop the receiver instantaneously, and then have the transmitter remove the data from the buffer and hold it, so that when the operation is restarted, the data is available for retransmission. Often, it is not necessary to extract items themselves for the buffer; all that

is needed is a count of how many are there. This is true if a copy of the transmitted items has been saved. For example, if the processor is transmitting to a device, the processor normally keeps a list of the items to be transmitted in memory, and moves a pointer down the list as a copy of each item is sent. In such a case all that is needed is a count of the number of items in the buffer, in order to back up the pointer. In contrast, a typewriter is an example of a device which has no copy, and would have to extract the items themselves from the buffer. In this case, the obvious question is what would the typewriter do with the items so extracted? In order to get a general answer to this question, consider another possible technique of dealing with the data in the buffer.

Perhaps the most simple thing to do with data is to leave it in the buffer. If in fact the I/O is just being suspended and will be restarted, why can't the items just remain in the buffer during the suspension? It turns out that this does not always work very well, as the example of the user's terminal will show. The usual reason to interrupt a computation is to modify it. Such a modification is often produced by the user at his terminal. Thus after the interruption of the process the next use of the terminal is not when the process is restarted but during the interruption itself. If data is left in buffers in anticipation of restarting, it will be in the way of any communication during the interruption.

What this example shows is that the terminal is particular, and many devices in general, are not dedicated to one task but are shared among the various tasks of the user. Whenever a device (or buffer or

process) is shared, any interruption of one task must be structured in such a way that another task can use that device. Thus going back to the earlier solution in which the typewriter extracted the contents of the buffer and retransmitted it when restarted, we see that this scheme is unacceptable because the sharing of the device means that the data so held will be in the way of other usage. The device could process data in this way only if it were very sophisticated, perhaps a small processor, so that it could implement and interface to the user a local buffer management strategy.

The structure of most systems is such that the processor is much better equipped than the device to cope with data returned in this fashion. Since a copy of the data usually exists, the state of the I/O operation can be represented by an index into the list; in order to multiplex the device it is only necessary to store this index.

In terms of the hardware implementation it is more difficult for the processor to remove the data in this case (writing) than it was in the other (reading) case, because the buffer is carrying the data away, not toward, the processor. If the processor is to remove the data, which given the limitation of the interface can only be done by read operations, then it will be necessary to reverse the direction of the data flow in the buffer so that a read operation will have the desired effect. Essentially what is needed is another buffer error recovery signal running from processor to device through the buffers, which, when set by the processor, causes the buffer to be forced into a state where the data can be reached.

If what is needed is the count of the items in the buffer, rather

than the items themselves, there is another possible implementation. Define a new aspect to the device, in addition to data, state word and record number, which is the number of items in the buffer. This requires that there be some module which is connected to both ends of the buffer, so that it can count items both as they enter and as they leave. Such a module is not consistent with the model of the buffers as a sequence of separate elements, but could easily be superimposed on that structure. If the buffer were implemented as one single unit, this function could easily be added.

In summary, four solutions have been proposed in order to deal with data left in a buffer by an interruption. 1) Back up to a check-point, if such exists. 2) Let the receiver of the data continue running to empty the data, if the receiver can run. 3) Have the transmitter get back the data or a count of it, if the receiver has some place to put it. 4) Leave the data in the buffer; this was rejected if the buffer was shared.

Which of the above four solutions is appropriate in a particular case is influenced by the very important consideration that in communication between computer and a human, the human and the computer have very different characteristics. In the previous example, the strategy of backing up to a check-point failed in the particular cases of communication between a computer and human: printer output, typewriter input and output. This is because the human reader is concerned with the printed form of the material he uses, and printed matter cannot be backed up. Similarly, the strategy of having the receiver recover items from the buffer and resend them later failed in the case of typewriter input, because

the typewriter was shared among various tasks. Actually, the user rather than the typewriter might be thought of as the proper agent to hold the items, but pushing characters back to the typist is, because of his concern with the printed form, not always possible.

The implementation of this buffering strategy has been presented in considerable detail, including state diagrams in Appendix A. The reasons for this detail are first, the desire to show that the characterization of the I/O interface as a number of memory words continued to be viable in the case of buffers. That is, it was necessary to show that using only read and write instructions, the processor can perform all of the functions which buffers imply. A proof by example seems the most simple approach.

Second, this implementation provides an example of the buffer model presented at the start of the chapter, in which each buffer element is executing an algorithm in parallel with the others. Since the buffer algorithms are usually (as in this case) rather simple and are fixed when the buffer is manufactured, the success of a buffer implementation lies in devising a buffer algorithm such that the correct overall behavior can be produced by reprogramming only the one algorithm executed by the processor. The algorithms presented here do not solve <u>all</u> I/O interface problems; they are particularly slanted toward devices such as typewriters, but subject to this restriction they deal with a variety of circumstances including abnormal halts. Thus they serve as an example of the sort of algorithms which will be required.

Again, it is important to stress that the thesis does <u>not</u> claim that this implementation of buffering is the only appropriate one. Buf-

fers could be a freestanding computer, or part of the device, as well as an array of elements. The important thing is the extent to which modification of the interface is required by the buffer. The insertion of the buffer caused several small changes to the normal operation of I/O. Short messages written using the wor strategy will need special start commands, and synchronization of state modification must be done explicitly. For error recovery, explicit attention to the buffer was required. In every case, however, the modifications did not preclude the use of the device interfaced as a number of memory words. That is the most important result of this example.

Other Forms of Buffering

If one looks at the sorts of buffers in use today, one can find structures significantly different from the buffer proposed here. An obvious question is how other sorts of buffers fit into this scheme. In order to discover the answer to this question, consider a different kind of buffer, similarly composed of a sequence of buffer elements, in which each buffer element does not transmit an item on its receipt, but holds items until it accumulates a certain number (a block) and then transmits this block to the next buffer as a unit. Clearly a buffer of this sort will have a different interface for inter-buffer transfer than it will for connection to device or processor. An example of this sort of structure is a "store and forward" message switching network.

One observation about such a network is that the buffers may be able to recover from certain errors without intervention of the processor. If the buffer keeps a copy of each block which it sends, and holds it

until the block is successfully received by the next element, then on failure of the transmission the block can be resent. This is actually an application of error recovery by backing up to a check-point, applied at the buffer to buffer level, rather than at the processor-device level. This observation is actually the first answer to the question posed above concerning fitting other sorts of buffers into the I/O scheme. In any I/O transaction, the operation can be viewed as going on at several levels simultaneously. The processor is trying to move items to a device, while a buffer is trying to move blocks to another buffer. The material developed in this chapter in terms of a one level transaction can and must be applied at every such level.

The strategy used to perform the operation at one particular level is often called a protocol. Thus in this case there is a processor-device protocol, a buffer-buffer protocol, and of course on the same level as the buffer-buffer protocol there are the processor-buffer and buffer-device protocols. Using this vocabulary, the buffering scheme developed earlier in this chapter attempted to make the processor-device protocol and the buffer-buffer protocol identical, so that the device could ignore the existence of buffers. This goal was achieved imperfectly. In a more complicated structure such as block transmission buffers, it is necessary to admit that there are at least two distinct levels of protocol. The general observations made so far about buffering will apply <u>to each level</u> of protocol in operation. For example, at any level at which error recovery can occur, the recovery procedure will follow one of the four techniques outlined in this chapter. Further, each level of protocol must be prepared to cope with errors which occur in

lower level protocols.

In the block-transfer strategy, error recovery at the buffer-buffer level was by backing up to a check-point. On the processor-device level, error recovery might be completely different. The technique of having the transmitter extract the data from the buffer might be appropriate, for example. At the processor-device level, the implementation details for these techniques might be quite different. For example, rather than the transmitter "turning the buffer around", the receiver might take the items out of the buffer and then resend them to the original transmitter. Another alternative would be to use the check-point technique by grouping the items into messages, each of which had a name, so that a check-point could be the beginning of the message. If it were necessary to retransmit from the point of interruption, that point could be described by the receiver to the transmitter in terms of offset within a message. From this example will come the other answer to our questions about fitting other sorts of buffers into our scheme: the same general technique will hold but the details of implementation will be quite different.

An Example of a Multi-Level Protocol

A good example of an I/O system which involves several levels of protocol is the ARPA communication network, a store-and-forward message switching facility (34) currently connecting over 40 computers, or hosts. The network which links these hosts is composed of interconnected Interface Message Processors, or IMPs, which could be described in terms of this thesis as block transfer buffer elements. There are several levels

of protocol in the network. At the lowest level there are the adjacent IMP-IMP protocol and the HOST-IMP protocol, which govern the transfer between adjacent modules. Above this there is the sender to receiver IMP-IMP protocol, between the IMPs which represent the ultimate source and destination of a message; above this there is the HOST-HOST protocol; and above the HOST-HOST protocol are special-purpose protocols for such things as transfer of files and allowing a user at one host to log into another host (the TELNET protocol). Details on these protocols are provided in several publications (1,28,29,39).

The adjacent IMP-IMP protocol is designed to allow for error detection and recovery. An IMP sends a block of data, called a packet, to its neighbor, and then waits for acknowledgement. The receiving IMP will send this acknowledgement back if the packet is received correctly. If on the other hand the packed is mal-formed, or if the check-sum maintained by the hardware indicates a lost bit, the receiver will do nothing. The sending IMP, on failing to receive an acknowledgement, will resend the packet, and will continue doing so until an acknowledgement comes back. This protocol, of course, is an example of error recovery by backing up to a check-point, the beginning of the packet. This protocol has no control lines such as ready-acknowledge which regulate the arrival of packets at an IMP. One IMP may send another a packet at any time without prior negotiation. If the receiving IMP is unprepared to accept the packet, it throws the packet away. The sending IMP will, of course, resend the packet when no acknowledgement returns.

The HOST-IMP protocol is very different from the adjacent IMP-IMP protocol. It has no mechanism such as check-sum for detecting errors in

the data transmission. Furthermore, bits are transferred across the interface one at a time rather than in packets, each transfer controlled by two signals somewhat resembling ready and acknowledge signals, except that the receiver rather than the sender of the bit must send the first signal.

No lines exist across this interface over which to report errors or request retransmission. Errors must be reported by sending an error message across the interface as if it were data. Such messages may originate in the local IMP, or they may come from a distant IMP as a result of the sender to receiver IMP-IMP protocol, if the message has been lost somewhere in the network. In this protocol as well, error recovery is by backing up to a check-point, in this case the beginning of the message, which is larger than a packet.

The sender to receiver IMP-IMP protocol and the HOST-HOST protocol deal with synchronizing inflow and outflow from the net, but the goal here is not cushioning delays but rather flow control: insuring that items do not come into the network faster than they are going out. In particular, input from one host must not disable other hosts' activities. To achieve this control, both these protocols require that before sending any messages the sender must obtain from the receiver permission to send a particular number of bits, and may not send more than this number.

In the case of an error, the sender to receiver IMP-IMP protocol contains a mechanism to cause the retransmission of a message. The HOST-HOST protocol, however, deals imperfectly with the need to remove information which is left in the system as a result of an error. In particular, the HOST-HOST protocol contains the concept of an event signal, the

meaning of which is to halt the process doing I/O, but the protocol fails to deal with data in transit when the signal is sent. The TELNET protocol, built on top of the HOST-HOST protocol, deals with such data by discarding it, a solution rejected by this thesis.

Thus the network control program in a host must deal with three levels of protocol. First, it must deal at the hardware level with the handshake-procedure necessary to transfer each bit to the adjacent IMP. Second, it must deal with the HOST-IMP messages, which do such things as report errors. Third, it must deal with the HOST-HOST messages which are concerned with multiplexing and flow control.

Summary

The insertion of buffers into the data path between device and device selector generates two problems, synchronization and error recovery. The intent of this chapter was to show that these problems could be solved in the context of the I/O interface which represents the device as a number of memory words. This has been shown by example, but in the process certain modifications were required of the interface between device and selector, and of the I/O control program.

The interface was modified by the addition of several new lines, the ror line and two lines for error recovery, the line which discards the buffer contents and the line which turns the buffer around. The revised interface is pictured in Figure 4-4. These lines do not change the basic nature of the device interface, however, but are rather additions to it.

The I/O control program must be modified so that it has explicit

knowledge of the buffer's existence. For example, the program must know how to recover from an abnormal halt which leaves items in the buffer. In general, it may be true, depending on the complexity of the buffer scheme, that the I/O program must be prepared to deal with two distinct levels of interface protocol, one level for the device and the other for the adjacent buffer. In this case the observations made in this chapter apply to every level of protocol.

The buffer scheme described here is an alternative to the frozen environment scheme of the last chapter, and was intended to handle devices which could not use the frozen environment. This buffer scheme is also capable of handling the record-oriented devices of the last chapter. The frozen environment scheme is felt to be the preferable of the two where it is applicable, however, for the complexity added to the control program by the frozen environment scheme, the addition of the system calls, is less than the complexity of error recovery and sychronization added by buffers. Thus the frozen environment will be used when possible.

Chapter 7 will propose a modification to the frozen environment scheme which will work with devices such as interactive terminals. Thus it might seem that buffers are of no use at all. In fact, buffers have a use perhaps more important than the one discussed in this chapter. Buffers were designed to allow devices to tolerate delays caused to the I/O control program. When the question of efficient use of processors is considered, a new delay will appear, the delay of processor scheduling. The buffer, of course, can deal with this delay as well as the delay caused by page exceptions. Thus the buffer scheme will appear in a later chapter.

Chapter 5

Multiplexing in the I/O System

Up to this point in the thesis, the assumption has been that there was no sharing among users of such things as I/O device controllers, communication lines, external buffers and other parts of the I/O machinery. In real I/O systems, these elements are often shared, or multiplexed. The purpose of this chapter is to introduce such multiplexing into this I/O system.

The term multiplexing will be taken to mean a sharing of some facility in such a fashion that the user could believe that the facility were exclusively his. In other words, this chapter will not consider arrangements for sharing which several users might work out, or which one user might work out for several of his tasks, although these arrangements are certainly practical and useful, but will rather consider schemes for sharing which the system may impose on the user. Such a scheme must not require the cooperation of the user, nor should it require the user to modify his programs to cope with it.

There are a variety of techniques lumped under the name of multiplexing. A good example of device controller multiplexing is the use of one set of control hardware to run several tape drives or disk spindles. An example of communication line multiplexing is the use of one cable to connect to the computer several typewriters at a remote location. Buffer multiplexing would reduce the cost of the buffering proposed in the previous chapter. Each of these kinds of multiplexing will be discussed in the chapter.

Multiplexing will be seen to cause a great deal of disruption to the orderly nature of an I/O system. Why include multiplexing at all? The reason is that great economies can be realized by sharing; multiplexing is currently necessary to achieve acceptable costs for I/O. This chapter, then, is an attempt to bring this I/O system closer to reality by adding multiplexing.

This chapter does not contain major new results. Rather, it is an attempt to evaluate various known multiplexing schemes in the context of this system. The evaluation will show that this system is compatable with various sorts of multiplexing, thus supporting the claim that this system is indeed an appropriate one; it will also show, hopefully, what aspects of multiplexing are the most disruptive to an orderly I/O system.

Sharing of the Ports on the Device Selector

The device selector was assumed to provide for each device a separate and distinct connection point, or port, and to map this port into a distinct set of memory addresses. Since it is by mapping portions of memory into the user's virtual address space that access to devices is granted or denied, it is crucial that each device be represented by a distinct portion of memory. If instead several devices were to share the same port, and thus the same portion of memory, it would not be possible to grant or deny access to these devices individually, which would mean that for protection reasons the user could not be given direct access to one of these devices, for in receiving access to one he would get access to all.

Why would one consider having several devices share a port on the device selector? The equivalent happens often in traditional I/O systems. If several remote typewriters are connected to the computer through a shared cable, that cable is normally connected to one port. Similarly, if several tape drives are run by one tape controller, that one controller is usually connected to one port In order that each device have its own port, it will be necessary, for example, that such a tape controller be connected to the selector by not just one cable, but by one cable for each device.

Thus the first conclusion about multiplexing is that whatever sorts of multiplexing are done should be done external to the device selector, so that each device is connected to its own device selector port. This conclusion is valid not just for this specific system. It follows from the basic assumptions implied by the goal of mapping devices into the user's environment in such a way that he can access them directly.

A Multiplexed Device Controller

In order to begin fitting multiplexing into this system, consider first the most simple case, which is device controller multiplexing. A tape controller running a number of tape drives would be an example. Since the controller is shared, it must move itself from one device to another to support the operation of the various devices. This section will consider what algorithm it can or should use to move from device to device.

One common strategy, called block multiplexing, is to assign the controller to a particular device for long enough to transfer an entire record or some other number of items. Let us first consider an alternative to this, which is that the controller moves itself from device to device on an item by item basis. Item by item multiplexing is perhaps a more natural technique to program, for as this chapter will show, item by item multiplexing can be done automatically, which eleminates the need for coping with whatever module performs the allocation in the block multiplexing case.

To understand item by item multiplexing, observe one particular device as it operates. At some point the controlling I/O program will issue a command to read or write. The multiplexor will then assign itself to the appropriate device and pass it the command. At this point the processor running the I/O program will wait until the device is ready to perform the data transfer. Clearly, the multiplexed controller must not remain assigned to this particular device during this waiting period, for it is the waiting period which consumes all the time which the processor wastes because it is faster than the device. If the controller were to remain assigned during the wait, it would be assigned to the device essentially all the time. So the controller must assign itself to a device twice during each transfer, once for the command and once for the data. The implications of this will be discussed below.

As more and more processes attempt to use the controller, it must share itself among more and more devices. This sharing implies that when a particular device is ready to use the controller, the controller may not be free. If the devices have no timing constraints, the only result

of this delay will be to cause the device to run slower. It is desirable, however, that the controller multiplex itself so that it gives each device a fair share.

If the devices have timing restrictions, it is possible that excessive delays injected by the multiplexing of the controller would prevent the device from being serviced in time. In this case it may be necessary to restrict the number of simultaneous users of the device. Block multiplexing can be viewed as the result of restricting the number of simultaneous users to one.

It is not hard to imagine an algorithm as part of the controller which gave a fair share to each device on an item by item basis. While it is similarly possible to create a controller strategy for sharing itself on a block basis, it is not clear that on a block basis the controller is the correct piece of the system to make this decision. If one process is to proceed, while others are halted, the other resources of these halted processes (e.g., pages in memory) ought to be freed. This suggests that the allocation of a block multiplexed controller should be coupled with the request which assures the user his other resources, the request to freeze the environment.

Doing block multiplexor assignments as part of the request for a frozen environment has another advantage in addition to committing resources only as needed, which is that the user then need not take explicit action because he is using a block multiplexed facility (provided that the system knows which devices are to be used during the freeze). Not introducing explicit action was one of the goals of

multiplexing. Thus the second conclusion about multiplexing which this chapter will draw is that in the context of this system a block multiplexed facility should be managed just as another resource, by committing it to a process as part of a frozen environment.

A Multiplexed Communication Line

Often, several terminals at a remote location share one cable to the main computer. The main difference between this sort of multiplexed facility and the multiplexed controller discussed above is that in this case the device and the communication line are not integrated in an arbitrary fashion as the device and the controller were. Rather, it is assumed that the device is equipped with the standard interface developed in the previous chapters, and the multiplexed line must connect to this interface.

It was observed in the previous section that under the item by item multiplexing technique, the multiplexor must assign itself twice in each transaction. In order for it to assign itself correctly in the second part of the transaction, the transfer of the data, it is necessary that the multiplexor be able to determine when the device is ready to do the transfer. In the case of the multiplexed controller this represented no problem, for the device and controller could be interconnected in any fashion necessary. The multiplexed communication line, however, must determine when the device is ready using only the information available at the standard interface. If the operation pending is reading, this is easy. When the device is ready to read, it signals over the

read ready line, which indicates that it is now appropriate to assign the line to this device. If the device is preparing a write operation, however, it will present no indication to the multiplexed line, for it generates no signal but rather waits for the write ready signal from the processor, which will not arrive until the line has been assigned. What is needed is a new line running from the device over which the device will signal when it is ready to perform the write. The multiplexor will, on receipt of this signal, assign itself to the device and pass the write ready and the data lines from the process on to the device. Since this line runs in the reverse direction from the normal write ready line, it will be called the reverse direction write ready line. The existence of this line will not alter the operation of device or buffer in any way except for generating the signal as appropriate. Figure 5-1 displays this addition to the interface.

The necessity of adding a new line to the interface does not result from details of the interface, but rather from two general observations. First, in any multiplexed facility such as this, which has "two ends", the device end and the selector end, it is preferable that one end only make assignment decisions, for if both can make them some additional mechanism is needed to prevent the various decisions from conflicting. This mechanism will require negotiation between the ends, which may be impossible in the case of a communication line due to restrictions of the line itself. The second general observation is that if one of the two ends provides the slower response of the two (processors wait for devices, not the other way) then that slower end (the device end) has the best assignment information available and should make the decision.

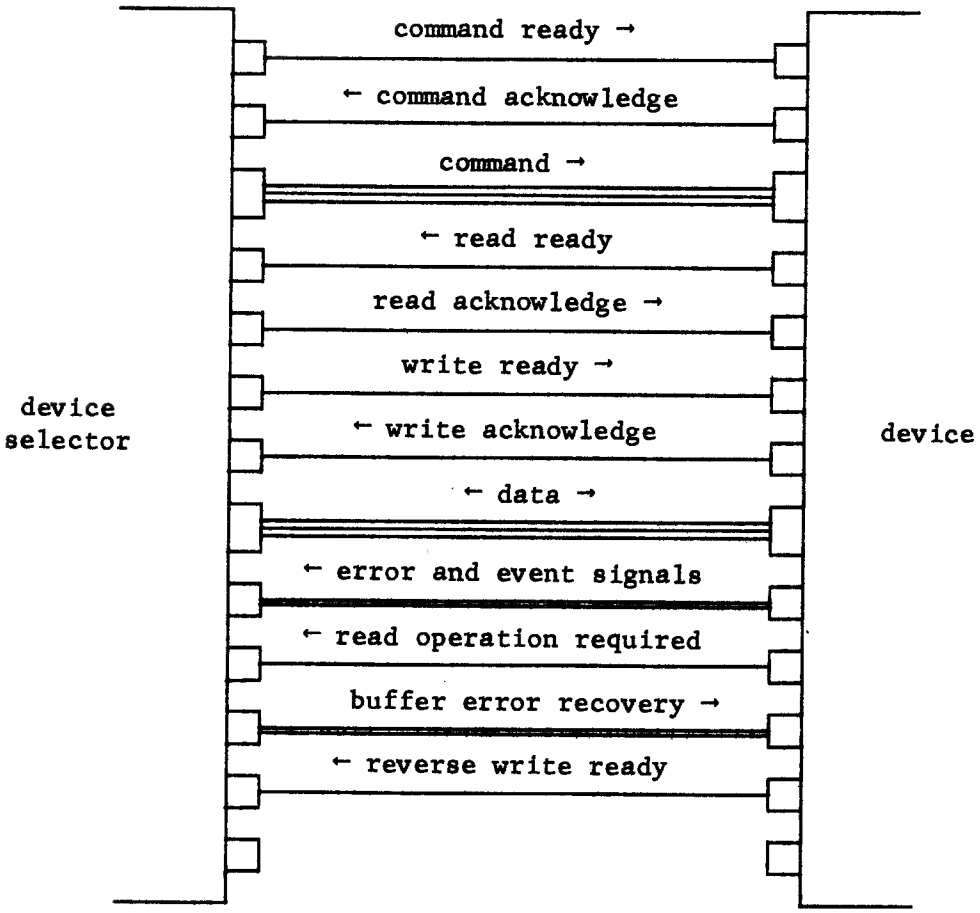

Figure 5-1: Device interface of Figure 4-4 with reverse write ready line added.

These two observations imply that the device must generate a signal whenever it is ready to complete a transaction. The necessity of localizing the multiplexing decision at the device end constitutes the third conclusion which this chapter will make about multiplexing, or about interface protocol in general.

There are facilities other than communication lines which might be multiplexed in the fashion, with two "ends" fanning out to devices and selector ports respectively. These observations would apply to any such facility.

Multiplexing of External Buffers

In the previous chapter, which discussed buffers, it was noted that while external buffers would cope with the various processor delays, for long delays or fast devices the amount of buffering required would be large. Rather than associate a large amount of buffering with each device full time, buffers could be assigned only as needed. This is multiplexing of external buffers.

Sometimes buffering is needed for the operation of another multiplexed facility. Asynchronous time-division multiplexing, often identified as ATDM, is the use of a multiplexed communication line assigned on an item by item basis with an insufficient limitation on the number of users to prevent the peak item arrival rate from exceeding the rate at which the line can handle the items. The benefit of operating in this mode is that the average number of items transmitted is increased, so that the line is more fully used, but in order to prevent the delays induced by the occasional peak arrival rates from causing timing failures,

buffering is needed. Rather than provide this buffering on a per-device basis, it is usually multiplexed as part of the ATDM scheme.

We will first propose, and find a flaw with, a simple-minded scheme for buffer multiplexing, which is to design a buffer element which resembles the one in the previous chapter except that it buffers additional information: the identity of the particular device with which the item is associated. When these elements are connected together to form a buffer, an item would go in one end accompanied by its device identification, and when it reaches the "other end" of the buffer the device identification would be sufficient to send it to the proper destination. Clearly, the buffer can only be used in one direction at once: if one device is reading, all must be reading, and this implies that there would be two buffers, one in each direction, for full operation. But this is not the flaw in the system.

Consider operation in one direction, say reading, and consider the end of the buffer adjacent to the device selector. There each buffered item in turn will be examined, its device identification extracted, and the item will be sent on its way to the proper process. Not exactly; the item at the head of the buffer will go on its way only when the program in charge of the associated device executes an instruction which picks the item up. But the user's program is not certified in any way. If it is badly written, or malicious, it may never remove the item from the buffer, in which case the whole buffer is stopped up and the scheme fails utterly.

What has gone wrong? The shared buffer, by putting the item of a user in its first element, is assigning itself to that user in such a way that the user must act before the buffer is free to assign itself to another user. This is a violation of the general rule that whenever an I/O program is not guaranteed by the system to behave in a certain manner, no multiplexed facility can allow its successful operation to be predicated on the co-operation of that I/O program.

One could attempt ad hoc solutions to this, for example a timer to limit the duration an item will be kept in the buffer. The real answer is that an item by item allocation scheme is inappropriate for buffer elements. If allocation were performed on a block basis, for example as part of obtaining a frozen environment, then the necessary controls on resource consumption would follow.

How might a block-allocated multiplexed buffer be designed. One simple configuration is pictured in Figure 5-2. In this particular scheme, the multiplexor has several buffers, one of which it will assign to a particular device as that device becomes active. If none are available, the process running the device must wait. Once one of the buffers is assigned to a device, it will behave as if it were dedicated.

More complicated multiplexed buffers can be imagined, which are capable of varying the amount of buffer allocated to each device depending on the dynamic need. A small computer might be used to run an external allocation algorithm which operated independently of the frozen environment.

The failure of the simple buffer multiplexing scheme is not a result of the details of this system, but is, once again, a general result of

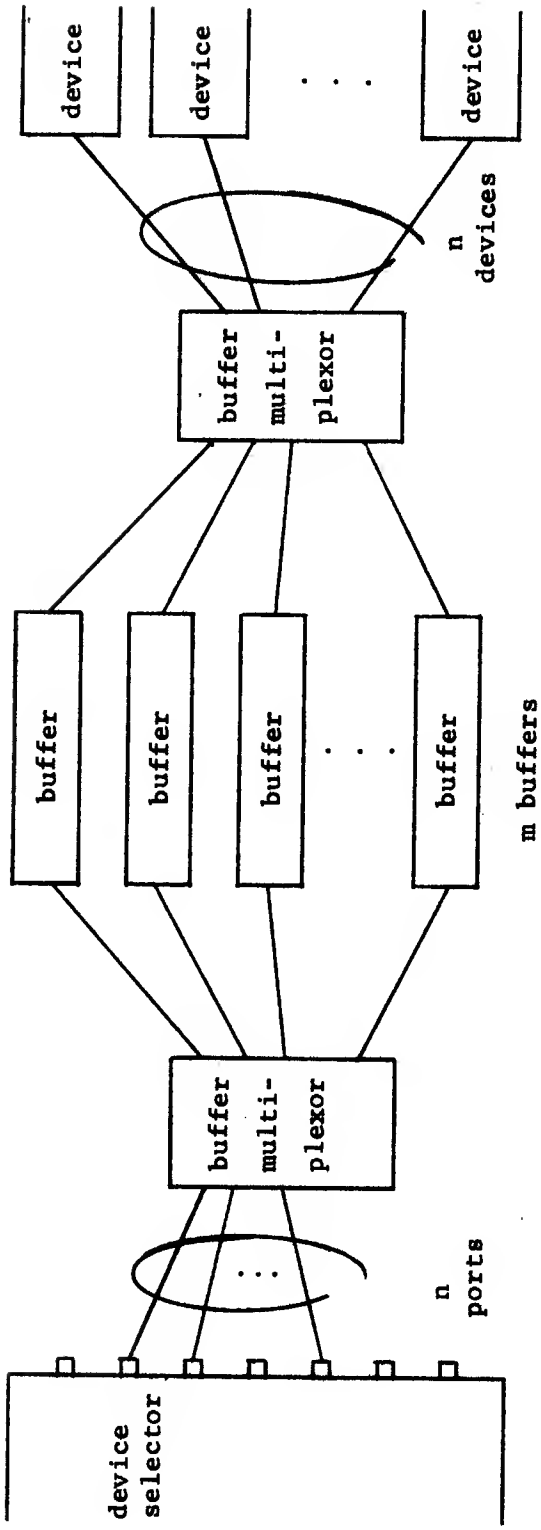

Figure 5-2: A scheme for multiplexing buffers.

the desire to allow the user to write and run an arbitrary I/O program. Allowing the user this freedom implies that anything the user can do must not disrupt any other user. Multiplexing is an obvious area in which this restriction will be felt.

While the use of multiplexed buffers does cause complexity, buffers dedicated to a particular device can be used in conjunction with other multiplexing without causing trouble. For example, it might be appropriate to buffer each typewriter connected through a multiplexed communication line. Such buffering would normally be connected between the device and the line, so that it could absorb delays in obtaining the line, as well as the other processing delays. In order that this work properly, it is only necessary that each buffer element have a reverse direction write ready line.

Multiplexed Ports Re-examined

As an earlier section discussed, the multiplexing of ports on the device selector is to be avoided because it would imply the loss of the ability to allow direct user access to the devices on the port. In a practical case, however, it may not always be possible to insist that each device have its separate port, for the system may have to cope with some multiplexed facility designed with the expectation that the main processor itself will demultiplex the information from the facility. An example of such a facility is the ARPA network, discussed in Chapter 4, in which messages from a large number of sites come through one port. In order to demultiplex this sort of facility externally, an additional computer would be required. In such a situation it may be necessary to

compromise and multiplex a port.

If this is done, the I/O control program which accesses the port must be provided by the system for protection reasons, and the user must access the devices on the port by communicating with this system certified I/O program. The important point about such a solution is that, assuming its limitations are accepted, it can be implemented easily in the context of this I/O system. The certified program which demultiplexes the facility will be run in an I/O process provided by the operating system rather than the user. User processes access the facility by interprocess communication, which is also true for an I/O process provided by the user. Thus the user, from his main computation, does not see access to these devices as being strongly different from other devices. And the use of one port in this fashion does not affect the operation of other ports. Thus in this system a multiplexed port does not cause a disruptive effect on other parts of the system; it may actually be a very appropriate strategy, if the resulting restrictions on that port are reasonable for the situation.

Summary

The I/O system developed in this thesis can be integrated with various sorts of I/O multiplexing. Multiplexing of a port on the device selector prevents the user from providing his own program to access the port, but other sorts of multiplexing, such as multiplexing of device controllers, communication lines, and buffers can be accomplished.

The main impact of this system on traditional multiplexing techniques is that giving the user direct access to the port implies that

the multiplexor must not allow faulty programming at one port to affect the operation of other ports. Multiplexed facilities must be more careful about port behavior than if the program using the port were certified.

The addition of the reverse direction write ready line was a reflection of the observation that in an interface protocol such as the one devised in this thesis, it is more orderly if the protocol for a class of transactions is always initiated on the same side of the interface, in this case the device side. The interface with the addition of this line is pictured in Figure 5-1.

In view of the various benefits which derive from the I/O system developed in this thesis, the restrictions which the system imposes on multiplexing seem worth the price.

Chapter 6

Processors as a Scarce Commodity

One of the major assumptions of this thesis has been that processors were inexpensive enough that one could be dedicated full time to any process doing I/O. The economics of today would make this assumption a rather expensive one; thus the goal of this chapter is to find ways of reducing the cost of dedicating processors full time to I/O.

There are two techniques which have been used to solve this problem, both of which can be made to apply to the I/O system being developed here. The first technique is to assign a processor to the I/O task only at the time when instructions are to be executed, and to assign the processor to something else whenever the I/O task waits for the device. In most systems which use this technique, the I/O programmer must explictily cope with the fact that his process is periodically removed from its processor. This chapter will show that in the context of this I/O system it is possible for the system to perform processor assignment automatically, so that an I/O program written under the assumption that it would have the processor full time need not be modified in order to use this technique.

The other technique which has been used to reduce the processor cost associated with I/O has been to transfer the I/O processing from the regular processor to a specialized processor, often called a channel, whose capability and cost are suited to I/O. This chapter will show that one of the important advantages of interfacing I/O devices as memory words is that a specialized I/O processor can be used in a much more versatile manner than in other I/O architectures. To the knowledge of the author, this versatility in the use of a specialized I/O processor has not been

exploited before in systems which represent the I/O device as a region of memory.

Dynamic Assignment of I/O Processors

The technique of assigning a processor only as needed usually finds application for typewriters and other slow devices, for the costs and benefits are well matched. In a traditional I/O system the technique might work as follows: whenever a typewriter needs to send or receive a character, it generates an interrupt. In response to this interrupt, the system runs an interrupt handler, a piece of code which takes the appropriate action, and then returns. How could this technique be fitted into the I/O system so far developed?

Clearly, one issue is efficiency. The switching of the processor to the I/O interrupt handler program must not be excessively costly, and must not cause so much of a delay that the I/O fails to be serviced in time. Many systems have demonstrated that this scheme can be made to run efficiently, but this thesis must consider whether efficiency is adversely affected by some feature of this I/O architecture, or by some feature of the sort of system in which it is embedded, for example virtual memories.

The other issue which this technique raises is one of program structure. Chapter 2 argued at some length that for reasons of clarity and ease of programming, the interruption of processes and the interrupt handler structure should be avoided. It will be necessary to devise some strategy for taking away and restoring the processor which does not destroy the structure of the I/O process, which so far has been sequential. Ideally, it would be possible to devise a scheme for taking away and restoring the processor

which is completely invisible to the user, so that he can create his I/O control program as if it were to be run on a dedicated processor. This, in fact, can be done.

The term suspension will be used to describe the act of taking away the processor from the I/O process. If suspension is to be done in a fashion invisible to the user, the system cannot ask the user for assistance in determining those points in the I/O control program where that program can be suspended. Instead, the system will have to identify these points on its own. But what are these points? They are exactly when the processor has attempted to reference an I/O device, and is waiting for the device to respond, for at those moments the processor is doing nothing, and can be put to some other purpose without affecting the I/O process.

Consider what state an I/O process is in when it waits for a device. The processor will have issued an instruction to read or write, the device selector will have issued the equivalent command, and the processor will then wait in the middle of the I/O instruction for the device to respond and the data to be transferred. It is when the process is in this state that it could be removed from its processor, and restored only when the device is ready to complete the transaction.

How could the conditions for suspension be detected by the system? An obvious way is with a timer. That is, if the processor makes a memory reference to a device selector, (rather than a memory box) and that reference is not completed within some time limit (a time of the magnitude of 10 memory cycle times, for example), and the request went to a device for which suspension were appropriate, then suspension could be initiated. This timer could be a part of the processor, which might in any case have

such a timer to allow recovery from a broken memory box which fails to respond, or it could be part of the device selector, so that the timer signal could be delivered only for appropriate devices.

How can the condition for restoration be detected? Clearly, the device must generate some signal which will cause the I/O process to be restored to a processor. How is this signal to be generated, and what is to respond to it? It is easy to generate the signal. When the device is ready to complete the transaction, it will signal over one of two lines in the interface. If the operation pending was reading, the device will signal over the read ready line. If the operation was writing, the device will signal over the reverse direction write ready line introduced in the last chapter. An obvious way to map these into a signal which will cause restoration of the processor is to define another line in the interface, to be called the need processor line, and adjust the interface so that a signal over the read ready or reverse direction write ready lines causes a signal on the need processor line. The need processor line is shown in Figure 6-1.

This need processor signal will be used as follows. When the need processor signal arrives at the device selector, the selector will in turn send a signal to some processor, the effect of which will be that a system routine is executed on that processor, whose function is to schedule the appropriate I/O process to run. Thus when the signal arrives, the I/O program is restored to a processor.

Suspension and restoration by this technique is essentially invisible to the I/O process. (Not completely, for example the quit handler process must be prepared to find the I/O process in a suspended state.) Such invisibility means that the program structure has not been disrupted by the technique, which was the desired goal.

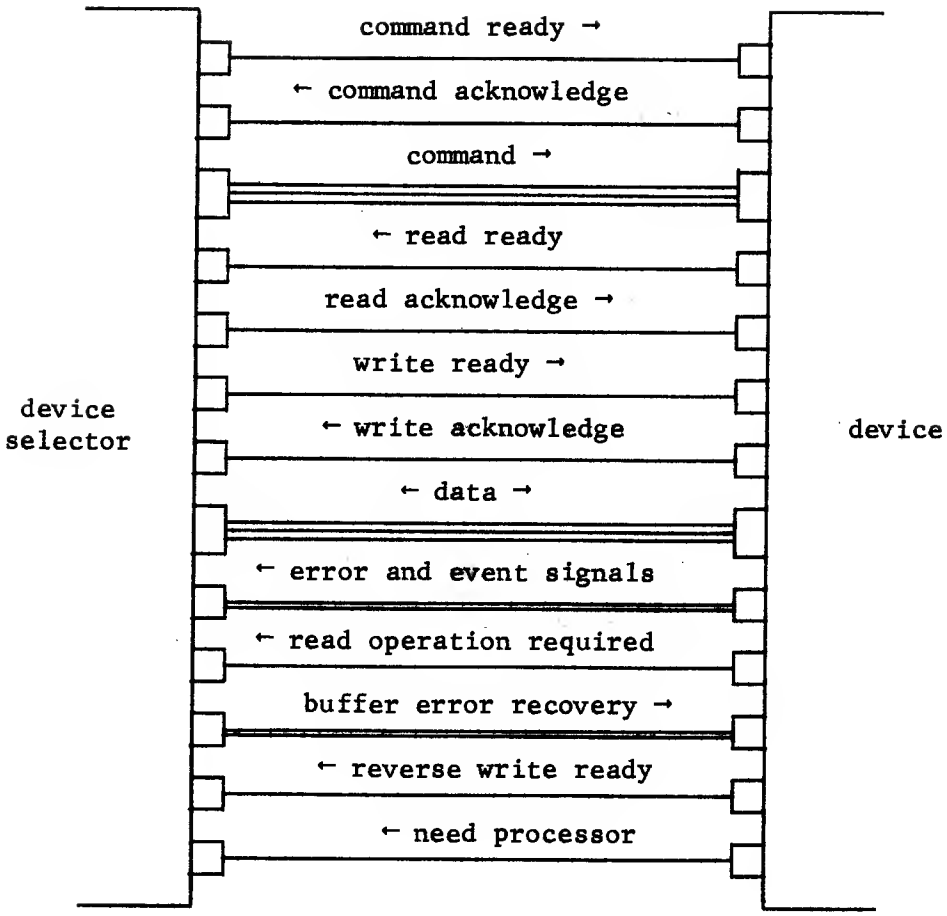

Figure 6-1: Device interface of Figure 5-1 with need processor line added.

This scheme for suspension and restoration shares some of the benefits and costs of paging. One benefit is the just mentioned invisibility, the fact that the mechanism is supported by the system so that the user need think little about it. One cost of this scheme, which has an analogy in paging, is the occasional inefficiency of suspending a process and being forced to restore it immediately. The equivalent cost of paging, removing a page from memory and having to fetch it back immediately, is tolerated as being outweighed by the benefits of paging: paging is automatic, performed by the system and hidden from the user. It would seem that the cost of suspension and immediate restoration would be similarly tolerated, especially since I/O is more predictable than paging, so the number of occurrences of this sort of inefficiency should be few. This analogy with paging was constructed to try to convince the reader that this scheme is viable even though it might sometimes do things which are not optimal. Without laboring the point further, let us pass on to other issues of efficiency.

One question of efficiency is will the cost of suspending and restoring the process and of holding the process in the suspended state be excessive. In many systems using interrupts and handlers it is not necessary to activate a complete process environment in responding to a signal. Thus this scheme might cost more. Some observations about an operational system may shed some light on this issue. In the Multics system, suspended processes are divided into two classes, called <u>loaded</u> and <u>unloaded</u>, depending on whether the system has kept in memory the tables for each process which maps the process virtual address space into the real memory. If these tables are loaded in memory, starting up a process is rapid, taking

about 300 instructions or about 1ms. Multics does such a scheduling each time the system takes a page fault. Measurements on the current system, with two Honeywell 645 processors, show that this scheduling can happen under peak load 100 times a second. If the process is unloaded, the cost of fetching these tables ups the cost and delay of restoration considerably, perhaps in the vicinity of 15ms. Thus it would seem that to implement this suspension scheme efficiently, all processes suspended for I/O should be loaded. If this were the case in Multics, typewriter input could be handled now on a per character basis, for measurements on the same system with 40 to 50 users show that the character input rate seldom peaks over 15 characters per second for the whole system, which would only add 15% to the loaded schedulings already done for paging. These additional schedulings would clearly not be unreasonable.

These schedulings are not now done in Multics because the memory to keep this many processes loaded costs too much. It was not a design goal of the Multics address mapping strategy to keep this many processes loaded, but as hardware costs decrease it does not seem at all unreasonable as a design goal for future systems. There are two techniques to reduce the cost of a loaded process. One is to restrict the generality of processes which are allowed to be I/O processes, i.e., to be kept loaded. A limitation on the number of segments (or a prohibition against making new ones while doing I/O) would fix the size of the necessary tables. The other approach is to remove any excess material from loaded tables. For example, in Multics, every address conversion table contains information about all the segments which compose the system supervisor itself, some 200 in number. This information is identical for all processes. If it were extracted and

placed in a common table, mapping tables thus specialized might shrink by an order of magnitude. This thesis has not developed a detailed virtual memory mapping scheme, so no context exists to pursue this issue in detail. Hopefully, the reader is convinced by these general observations, and also by the falling prices of memory, that it is not unreasonable to keep I/O processes loaded. This topic is really the efficient use of memory, not processor, and will be discussed in the next chapter. Let us therefore leave it, for there is more to say about scheduling efficiency.

Can the device tolerate the delay in completing the transaction caused by restoring the I/O process? Obviously, this depends on the device. Some devices, once they are ready to complete the transaction, will tolerate little further delay. Others will tolerate any amount of delay. Clearly, the tolerable delay is a limitation of this scheme. The other limitation is the overhead of the system scheduling routines, which will take more and more resources the more often the device transfers items. To circumvent these limitations, some mechanism is needed which will allow more to be done on each scheduling, so as to reduce the frequency of scheduling, and which will help cope with the delay in processor scheduling. Does such a mechanism present itself? The external buffer of Chapter 4 will work quite well.

Buffers as a Tool for Processor Scheduling

It is easy to show that the external buffer of Chapter 4 will increase the amount done at each scheduling, and will cushion the scheduling delay. Clearly, the buffer can cushion the scheduling delay, for the delay in scheduling a process is no different than any other delay with which the

buffer was designed to cope. In order to see how the buffer provides more work at each scheduling, consider reading as an example. If, before the processor is restored, the device is allowed to fill the buffer up, then the processor, when restored, can read not just one item, but all the items in the buffer. Obviously, it is not appropriate to let the buffer fill completely up, for then there would be no room left to cushion delays in scheduling. But sufficient buffering can be provided to allow a certain number of items to accumulate.

In order to delay restoration until the buffer is partially full, it is only necessary to signal over the need processor line when the buffer is appropriately full. A clean way to implement this is to propagate the need processor line through the buffers so that any buffer can turn it on, and then provide a modified buffer element which will signal over the line when it changes from the empty to the full state. By positioning this element properly in the middle of the buffer, the signal can be generated when the buffer is filled to any desired degree. This simple modification is all that is required to make the buffer work as a tool in processor scheduling.

Clearly, this scheme works for writing as well as reading; the only difference is that the device empties rather than fills the buffer, so that a buffer becoming empty rather than full must trigger the need processor line.

What are the limitations of the buffer scheme? First, it does not avoid but only puts off the issue of efficiency. This scheme decreases the scheduling cost associated with a given transmission rate, but does so only by use of a larger and larger buffer. Clearly, when the cost of the buffers

equals the cost from excessive scheduling of processors, the scheme has no further use. Moreover, as was noted in the last chapter, buffers are one of the most difficult items to multiplex, so it might in fact be necessary to provide this large amount of buffer separately for each device, which seems especially inefficient.

The other limitation of the buffer is the previously discussed fact that the data flow algorithm in the buffer is fixed and built in as part of the buffer. The inflexibility of the algorithm is a special hindrance when the buffer is used to help schedule the processor, for the reason that for certain kinds of devices (typewriter input is the best example) certain computations must be done on each item as it arrives. Typewriter input may require checking for the arrival of a control character, echoing a character (or a sequence of characters if any sort of automatic typing completion is implemented), and checking for characters which indicate the end of message. The use of buffers postpones these computations until the processor is scheduled. In the previous chapter, the processor was always trying to catch up with the buffer, and the only postponement was caused by delays such as paging. Now, with buffers serving as a scheduling tool, the postponement is of a different degree; it lasts until the processor is scheduled, and scheduling will only happen when something causes it. What will cause it? Not the buffer. It only knows to signal for process scheduling when it is filled up to a certain amount, and that is not the right criterion. One character along might arrive, which should get immediate processing, but if it doesn't fill up the buffer, the character could sit forever. Clearly, to avoid the delay, these sorts of computations would

better be done at the device end of the buffer. But the buffer cannot do this, for that part of the algorithm is fixed.

There are three solutions to this problem, two ad hoc and one general. The rest of this chapter will deal with the general solution, but the two ad hoc solutions ought to be mentioned in passing. First, if the computations which need to be done at the device end of the buffer can be determined in advance, special boxes can be built which will do them. These boxes could then be spliced in between the device and the buffer. The box could either do the computation or detect that it needed to be done and signal on the need processor line. The limitation of this scheme is obvious: the algorithms needed must be correctly predicted in advance. Any error is awkward and expensive, for once again the boxes with their algorithms are prefabricated and not programmable. The other ad hoc scheme is to set a timer which goes off periodically and causes a processor scheduling. In order to provide reasonable response, the timer should probably go off fairly often, perhaps every few seconds, and this in turn implies that to avoid hopeless inefficiency the timer should go off only if something is in the buffer. Otherwise the process of a logged in user would be scheduled every few seconds for the whole console session. Even with this condition on the timer, the scheme provides an upper limit on efficiency, at the same time providing an upper limit on response. If a suitable value of the timer interval can be found to satisfy both criteria, the scheme can be made to work. But there is no guarantee of success in any particular case. Clearly what is needed is some programmable module less expensive than a processor, which can execute I/O programs. This is the topic of the next section.

A Specialized I/O Processor

Traditional systems have often used a specialized I/O processor to execute some parts of the I/O control program. Such a processor is often called a channel. The term "channel", however, has many different meanings, so this thesis will not use it, but will instead use the term specialized processor, or SP, by which will be meant any processor the characteristics of which have been tailored for executing the I/O program. Perhaps the SP can be scheduled with less overhead, so that it is especially appropriate for the technique of suspension and restoration. Or perhaps the SP might be a very inexpensive version of the main processor, perhaps lacking a cache or some fancy machine instruction, so that minimal cost is incurred if it sits idle. This section will explore how, in the context of this I/O system, a processor might be specialized for I/O.

Structurally, what is the difference between a specialized processor and a traditional channel? In traditional I/O systems in which all I/O is done through channels, one important role of the channel has been to provide the connecting point for the devices. Obviously, SP's will not serve that role in this thesis, for the device selector has that function. Thus in this system a SP is nothing more than an alternative form of a processor, differing in cost and capability, but identical to a processor in the manner of its interface to the rest of the system modules. That is, an SP, like any processor, will do I/O by making memory-to-memory moves. And like a processor, an SP is not attached to one device, but may be used to operate any device (or combination of devices). Structurally, then, an SP is rather different from a channel in a traditional I/O architecture. Compare Figure 6-2, which shows an SP added to this system, with Figure 2-1, which shows a traditional I/O processor.

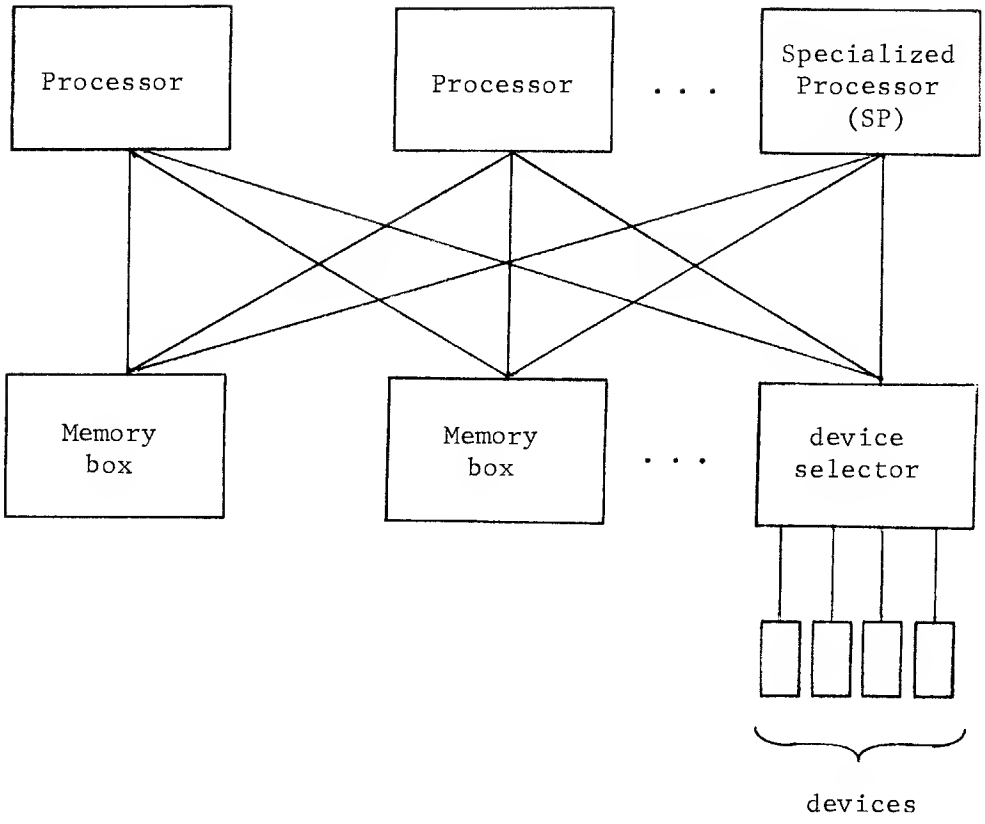

Figure 6-2: I/O system augmented by addition of specialized processor.

This structural difference is an important advantage of the I/O scheme of this thesis. By taking the traditional role of the channel, and splitting it into two parts, one of which is performed by the device selector, the scheme eliminates many restrictions imposed on SPs. Fewer SPs are needed, since an idle device does not imply an idle SP. Rather, an SP will be assigned to operate the device only as needed. Similarly, the failure of a particular SP does not mean that some particular device is inoperable until the SP is fixed. The system can just operate with one less SP during repairs.

What sort of criteria will define a successful SP design? Two conclusions from previous chapters seem especially relevant. First, to this point I/O has been programmed using the normal computer language. This is a desirable feature, so hopefully SPs can be designed which execute a language similar to that of the main processor. The other conclusion is that the primitive communication mechanisms, e.g., interrupts, which often are used between channel and processor can lead to awkward program structure. Hopefully, SPs can be designed which do not disrupt the sequential nature of I/O programs.

The previous paragraphs have listed some constraints on SP design, two goals: minimum cost and maximum flexibility, and two bad features to avoid: special languages and awkward program structures. Subject to these constraints, what sort of SP might be built for this system? Before proceeding, let the reader be warned that this thesis will not present the detailed design for a particular SP. A detailed design is inappropriate because there are several alternatives, depending on relative merits and costs of the particular system. To pick among them here would serve no

useful purpose. Rather, this section will discuss various design techniques which seem appropriate.

The following is a list of various techniques which might be used to produce a less costly version of a processor to use for I/O.

Speed reduction: processors of today often employ sophisticated techniques to increase speed, such as instruction stream pipelining or caches. This speed may be unnecessary for I/O, which is often much slower than the processor.

Elimination of special instructions: hopefully the SP and processor will have similar instructions sets. One difference which might make sense would be the elimination of certain complex instructions unrelated to I/O. Special decimal arithmetic instructions, for example, might be omitted from a SP.

Sharing of hardware among SPs: certain functions which are needed by several SPs can be implemented as a single module which is shared among them. The effect might be a loss in SP speed, but this is not necessarily bad. Examples of functions which might be shared include the support of certain special instructions, and the conversion of virtual to real addresses.

The last technique, sharing, suggests an alternative approach to reducing the cost of SPs. One limit of the technique of suspension and restoration was the overhead of switching. Sharing of processor parts is an alternative technique for giving processing power to a process only as needed. By pushing the idea of sharing to its limit and building a processor which, by sharing all its components among several processes, tried to multiplex itself to produce the effect of several processors, one could perhaps produce a variant of suspension, implemented by the hardware

itself, which was less expensive, or more rapid, than suspension of regular processors.

Thus there seem to be two techniques for the production of inexpensive SPs. Make the SP so cheap that the cost of letting it idle is negligible, or make the SP multiplex itself very efficiently between several processes.

Multics, for example, contains a hardware module, the generalized I/O controller, or GIOC (32) which attempts to provide inexpensive processors to perform I/O, by sharing modules among the several processors. Examples of shared functions include the module which references memory and the module which increments the instruction counter and picks up the next instruction for the I/O program. For slow devices, the GIOC even shares the live registers which the control programs use. In many respects, the processors provided by the GIOC differ strongly from SPs, and rather resemble traditional channels. They execute a special language having nothing to do with the central processor language, they provide the connection point for the device, and they communicate using interrupts. The idea of cost reduction by sharing is, however, well demonstrated.

The next section of the chapter will discuss other implications of SPs. Before going on, it seems appropriate to pause and review the role of SPs and the various other techniques for controlling processor utilization. Four techniques have been identified: suspension and restoration, appropriate for very slow devices, suspension and restoration with buffers, appropriate for medium speed devices where no special computation is needed, SPs, appropriate when programming flexibility is needed or buffers would be too large, and lastly, the technique of the previous chapter, use of the processor itself. Initial reaction may be that this last technique is

never appropriate by today's economics, but in the Multics system today there are devices which transfer so rapidly that, even though the transfer is performed by a channel, no other use can be made of the processor during the transfer, for there is not time even to start another task running on it. Clearly, for this sort of device the channel is superfluous. There is another class of I/O for which the use of the processor itself may be appropriate; this is a very infrequent transaction. If a device is used very seldom, it may be worth the cost of inefficient use of processor during its operation to avoid the cost of maintaining special mechanisms, such as dedicated buffers. As processors get cheaper, which seems to be the current trend, this technique will have a wider and wider range of applicability.

It is also appropriate to discuss the economic practicality of the SPs here proposed, for clearly they are complicated compared to the channels to be found in existing systems. First, it is very important that SPs can be shared from device to device, for fewer of them are thus required, which raises the allowable maximum cost. Second, the cost of hardware is coming down. If the regular processors can be used for really fast transfers so that the speed required of SPs is not excessive, then various current fabrication techniques such as micro-programming may allow the production of sophisticated SPs very cheaply. For these reasons we have no hesitation in saying that for these circumstances in which SPs are required, it would be feasible to provide them.

Program Structure Induced by SPs

This section will show that the execution of some portions of the I/O program by a specialized processor can not fail to have an effect on the

form of the I/O program. However, this effect can be minimized, so that undesirable program structures such as the interrupt handler structure do not occur.

It is possible to imagine that in general the programs which execute on processor and on SP must be viewed as two processes executing in parallel. In fact this view is appropriate under certain circumstances, but it certainly embodies a considerable complexity, so let us begin with the most simple case, which is that the I/O program is written as if it were to run just on the processor. Without other modification, this program could be broken up into various parts, some of which execute on the SP. A scheme to do this which used "switch-processor" instructions inserted at appropriate points was proposed by Smith (37). That is, the switching from processor to SP and back was programmed explicitly by the user. It seems that it is necessary to perform this move explicitly, for no criterion can be stated which will enable the system to determine when a process should change processors. Actually, it is easy for the system to tell when the I/O process should be moved from processor to SP; the conditions are exactly the same as those which triggered suspension: excessive delay in the device's response to a read or write request. But when is the process to be moved back to the processor? Clearly, when the I/O transaction is finished. But there is no way for the system to detect this. It seems simpler to let the user decide what portions of his code should be run on which processor.

Under what circumstances would a more complicated program structure be required? When would it be necessary to view the program on the processor and the program on the SP as two processes, executing at the same time? Exactly when the SP, because of the steps taken to simplify it, lacks the

power to execute some computation which must be done by the I/O control program, and in addition, the time which would be spent switching to a processor to do the computation and then back to the SP would cause such a delay that the I/O program might fail to meet timing constraints. It is possible to claim that the occurrence of this situation indicates that the SP was improperly constructed, but issues of economics might force this undesirable situation to hold. Certainly for channels found on current hardware, where even a simple computation such as sending a message to another process is beyond the capability of a channel, it is necessary to invoke the main processor in parallel with the channel. In this case the simple "switch-processor" instruction must be replaced with something more complex. An appropriate mechanism is described in the next section.

A Channel-Processor Programming Scheme

This section describes a specific mechanism which was devised on Multics to deal with the coordination involved when the processor and the SP must operate in parallel. Since, as the previous section pointed out, the necessity for this coordination can be avoided by proper SP design, the description of this mechanism is not crucial to the thesis. It has been included to make the following point. The thesis has argued the advantages of eliminating interrupt driven programs and replacing them with programs having sequential structure. Even if the reader has been convinced in the abstract, he may think that in practice he cannot exploit this beneficial structure, because he is constrained by his hardware, which uses channels and interrupts. The purpose of this example is to present a scheme which,

in the context of channels and interrupts, allows the user to construct I/O control programs which appear to be sequential in nature, and which mask the existence of interrupts.

The scheme to be described is an idealized version of the I/O control, or IOC language, which was developed and implemented by Stanley Dunten on the Multics system for the control of typewriters. As stated above, in the Multics system all I/O is under the control of channels, and these channels communicate with the main processor by means of interrupts. Associated with each interrupt (in this reather idealized version of IOC) is a short message, the interrupt index, which is one of the integers between zero and some small number. (Numbers less than ten would be sufficient for all the currently written programs.) This interrupt index is used by the run-time environment of IOC, as will be shown.

The following is a description of the typical sequence of operations which one would use in the IOC language to start a channel program, and to coordinate pieces of processor code with it. The programmer would first code the channel program itself, in line in the IOC program. This program, in the special channel language, would specify, among other things, the particular interrupt index to be returned at each point where the channel program might generate an interrupt. Following the instructions for the channel in the IOC program will be instructions to be executed by the processor. The first of these will normally be a wait instruction, the effect of which is to put the process in a wait state where it abandons the processor pending an interrupt. The interrupt index is used as follows. Part of the wait instruction is an array of labels. When an interrupt arrives, if the process is in the wait state, a transfer is executed to the label

selected in the array by the interrupt index. If the process is not in the wait state, the interrupt will be queued until such time as it is.

The code which is executed by transferring to one of these labels will be of two sorts. If it represents some computation which the processor should do in parallel with the running channel, then the sequence of code, when it finishes, must return to the wait state so that further interrupts can be accepted. To do this the code ends either with a new wait instruction or with a waitagain instruction, which returns to the previous wait state. Alternatively, if the IOC program does not execute a wait or waitagain instruction, it is assumed that control has returned permanently from the channel, and that after sending the interrupt the channel program has halted.

Signals may be received from two sources other than the channel: from the timer, and from other processes in the user's computation. These signals, just as those from the channel, can be received only while the process is in the wait state, and these other signals are made to have a syntax similar to the channel signal by mapping them into special interrupt indexes which cannot be generated by the hardware. These other signals are used as follows. Normally a timer is started whenever a channel program is started. If the channel fails to operate properly, and no interrupts are generated, the signal from the timer prevents the process from waiting forever. The signals from other processes are the means by which those other processes make requests of the I/O process. In Multics there are four requests, whose meaning is as follows. 1) Abort any writing in progress. 2) Start writing (the data to be written is in a shared area). 3) Hang up the

terminal and destroy the I/O process. 4) Update the count of input characters typed. These are the only signals which the I/O program can receive.

The significance of this program structure is that although the channel generates an interrupt, the IOC program does not see it as an interrupt, but rather as a signal which arrives only at the points at which the program is prepared to receive it: at wait instructions. Thus using a channel with only limited expressive ability, the wait instruction produces an interprocess communication with the property discussed in Chapter 2 that the signal arrives only when process has placed itself in a state where it is prepared to accept it. The I/O program is, in the literal sense of the word, never interrupted. If the reader will remember what in Chapter 2 was called the interrupt handler program structure, he will see that the awkwardness of such a structure has been avoided. Thus, even in the case of limited channel capabilities and excessive overhead in process switching, sequential program structure can be achieved.

Impact of Process Suspension on Multiplexing

In order to complete this development of processor sharing, it is necessary to discuss the impact of process suspension on the material presented in earlier chapters. This section discusses the modification to the multiplexing scheme of Chapter 5 which is implied by the scheme of process suspension and restoration. One assumption of that chapter has been invalidated: that when the device is ready for the transaction, the processor will always be ready. This assumption was used in deciding when to allocate a multiplexed controller or line to a particular device in that the controller

tested only the readiness of the device. Now it would seem necessary to test both device and processor to make sure both are ready before assigning a multiplexed module, in order to avoid having the module sit idle while the processor is scheduled. This, in the case of a multiplexed communication line, requires negotiation back and forth between the ends of a multiplexed line, which is not always possible within the design of the line.

There is an alternative to this, which is to assign the multiplexed line as before, whenever the device is ready, but to observe that the line, like a device, has a time limit within which the transaction must be completed. If, because of delays due to process restoration, the time limit cannot be met, then, as before, buffers can be used to absorb the delay. This is a different use of buffers than the one discussed in the chapter on multiplexing. That chapter considered placing buffers between a multiplexed communication line and a device. This problem requires that the buffer be placed next to the processor.

How would this buffering work? Consider reading. When the device has an item ready, it will signal over the read ready line, and at some point will be assigned the communication line. Assuming that the buffer is empty, the item will be transferred immediately into the buffer, and the line will be relinquished. At some later point, the processor will remove the item from the buffer. Now in order that this work properly, the device must never signal over the read ready line while the buffer is full. But a quick review of the buffer algorithm will convince the reader that this signal can never happen, for the buffer will never pass a read data command on to the device so long as it contains an item, and until the device receives a read

data command, it will not give a read ready signal. The interested reader may wish to convince himself that writes work as well.

Summary

The goal of this chapter is to reduce the cost which results from dedicating a process full time to I/O. Four techniques are identified.

The first technique, and the most simple, is to observe that for certain kinds of devices (very fast devices and infrequently used devices), the cost of the scheme of the earlier chapter, in which the processor itself was used, is not as great as might be imagined. Observations about the decreasing cost of hardware are used to support this approach.

For very slow devices, the chapter discusses the technique of suspension and restoration, in which the process is assigned to a processor only when program execution is required. This technique required the addition of one line, the need processor line, to the interface.

For those cases in which this technique causes too much scheduling overhead, the external buffers of Chapter 4 can be used to extend the number of operations done on each scheduling, and to cushion the device against scheduling delay.

The fourth and final technique is the use of a specialized processor or SP, useful if the fixed algorithm of the buffer is insufficient or the amount of buffer needed is excessive.

The chapter does not propose a specific SP design, but discusses several design criteria. It proposes two goals, maximum programming flexibility and minimum cost, and two features to avoid, creation of a special programming language and introduction of an awkward program structure.

The chapter discusses in some detail the effect of SPs on program structure, and concludes that while SPs will have some effect on the structure of the I/O program, the awkwardness of the interrupt handler structure can be avoided. Indeed, the chapter shows by example that desirable program structure can be produced in a less than ideal I/O environment.

By interfacing devices as memory addresses, the function of providing a system interface for the device, which is often the task of the channel in traditional I/O architecture, is separated from the functions of the I/O processor and is made the task of the device selector. The simplification of the SP which results from this separation is very important, in that SPs thus perform I/O by executing memory-to-memory moves rather than special I/O instructions. Further, SPs are no longer fixed to a particular device, but can be used on any device, as needed.

Not only can an SP be moved from device to device, but any device can be referenced equally well by any of the SPs or main processors. For a device which operated at more than one speed, for example, the user might write an I/O program which referenced the device sometimes using a SP and sometimes using the processor itself. Or the user might experiment with some new device using the processor, and then write a final program, more efficient but more complex, which used a SP. This flexibility would vanish if the device were to be connected to any specific processing element, e.g., a channel or an I/O bus which is connected to some register in a processor. Only in this chapter, when the possibility is raised of more than one kind of processor, does this very large advantage of the device selector as a separate entity become apparent.

Chapter 7

Memory as a Scarce Commodity

The last chapter considered ways of reducing the processor cost of I/O. This chapter will consider memory in a similar way, discussing techniques for reducing the committment of memory required for I/O.

In Chapter 3, the discussion of memory management listed three goals which a particular allocation scheme must meet. They were first, that the allocation of memory to I/O must not prevent other tasks of the system (the example was reconfiguration of memory) from operating properly. Second, memory must be allocated in such a way that no user can get more than his fair share. Third and last, the allocation of memory must be done in such a way that the cost to the user of doing I/O is not out of proportion to other costs in the system. The frozen environment scheme using a fixed time limit met the first two goals, but did not attempt to deal with the third. Rather it assumed that the costs associated with using paging as the storage allocation scheme for I/O would be acceptable. Unfortunately, examination of a system such as Multics will reveal that by today's standards the cost would be unacceptably high. (It is assumed that the price the user pays does reflect true cost.)

This chapter will reduce memory costs associated with I/O by introducing an alternative to paging which allocates memory in a manner more suited to the characteristics of I/O. It will then show that by introducing this more efficient scheme, the groundwork has been laid for a variety of improvements to the frozen environment scheme which will allow additional sorts of devices, including typewriters and interactive terminals, to take advantage of the scheme.

Enlarging the class of devices which can use the frozen environment scheme becomes more important with the introduction in the last chapter of specialized processors, or SPs. The introduction of SPs as an alternative to processors did not change the requirements which the I/O makes of memory. The SP still must either run in a frozen environment or in conjunction with buffers. The last chapter observed that the use of SPs will force a modification of the program structure to at least a small extent. It would be very nice if the programmer coping with SPs did not at the same time have to cope with the complexity of buffers. The alternative of the frozen environment is not acceptable, however, unless the I/O is capable of running within the constraint of a fixed time limit, which is not always the case. An even stronger reason for wishing to use a frozen environment with SPs arises if the SP has reduced functional capability compared to a processor. It is possible that in this case the SP does not have the capability to handle a page exception. In this case, when the SP must request another processor to fetch missing pages, the ability to avoid page exceptions by use of the frozen environment is especially important.

This chapter, then, has two objectives. First, reduce the cost associated with frozen memory, and second, increase the kinds of devices which can take advantage of the frozen environment scheme. As we show that achievement of the first objective is central to achievement of the second, we will show that memory costs are the cause of a variety of compromises and restrictions which are part of many I/O systems, as indeed they were a part of the first frozen environment scheme.

Memory Costs Associated with I/O

One of the assumptions which was made early in the thesis was that the system in which the I/O was to operate used paging as its memory allocation scheme. The use of this allocation scheme for the memory which is involved in I/O causes the high cost associated with using the frozen environment scheme in a Multics-like system. The reason is that the page sizes commonly used (Multics currently uses 1024 words per page) are very large compared to the storage needed for I/O. An example of typewriters in Multics will show how much memory is saved if only the storage needed is frozen in memory, rather than all of the page containing that storage. A running typewriter in Multics requires three storage areas in memory, the area holding the I/O program, the area holding the data items, and an area for program variables. Each of these is less than sixteen words (there may be more than one data area). If holding each of these areas in memory required a full page, the storage consumed would jump from 48 words to 3072 words. If the areas could be arranged in the same page, they could still use 1024 words. The increased cost of these extra words is sufficient to deter the use of the straightforward frozen environment scheme, even if the other two goals could be met.

Obviously, in the Multics case, in order to achieve acceptable cost, paging has been abandoned for typewriters in favor of a specialized memory management technique which allows very small blocks of storage to stay in memory efficiently. In fact, some specialized management strategy for storage related to I/O is used for every device in the Multics system. The next sections will try to integrate a special I/O

storage management strategy into the system in such a way that the good features of the I/O system are not disrupted.

Cost Reduction through Memory Management

The techniques used in Multics to implement special management strategies have certain drawbacks. The normal technique is to write a special storage manager for use with each particular device, which obtains a segment from the virtual memory manager, and then implements its management algorithm within the segment. The disadvantage of this technique is that along with the management algorithm the manager must implement the protection strategy which controls the access to this specially managed storage. Clearly the access controls of the virtual memory manager will not help; they protect the segment as a whole, not the areas allocated within it. Lacking these access controls, the special storage manager has no alternative but to deny the user direct access to the managed area. The result, of course, is that the user cannot write his own I/O program, but must rely on system software to perform his I/O.

In order that the user to allowed direct access to the specially managed area, the access controls of the virtual memory must be used to regulate the area, which means that the special management strategy must be implemented by the virtual memory manager itself, and not some other module.

In the introduction to the thesis it was observed that I/O would become tractable when the similarities rather than the differences between devices were identified and exploited. This is a good example. The virtual

memory manager cannot be expected to implement a different strategy for each device. It will be necessary to sacrifice a little of the efficiency of memory usage and identify a common strategy which many devices can use, before the benefits of management by the virtual memory can be obtained.

To see what sort of management strategy would be appropriate for I/O, look again at Multics. The various storage areas required in core vary in size but are often very small. Many of the I/O control programs are between ten and twenty words. Auxiliary areas for variables are often similar in size. The data items themselves require storage areas of various size, ranging upward from 16 words. (Some of the larger areas are the size they are in order to interact well with the paging mechanism, rather than from device constraints.) Another observation is that none of these areas ever change size. Under certain circumstances the effect of growth is simulated by using a greater or fewer number of areas, but in no case does the size of areas change.

One could try to improve the efficiency of memory utilization by using a very small page size. This reduces the number of extra words which fill the rest of the page, but requires the use of a page table which for small pages gets more and more wasteful, since it uses up one entry for each page, regardless of page size. More importantly, the use of small pages increases the overhead associated with moving pages in and out of primary memory, since the cost of keeping track of the pages in memory, selecting pages for removal, preparing the control program for the secondary storage device, and so on, is independent of page size.

An alternative memory allocation strategy which works well for small segments of fixed size is contiguous allocation of these areas in memory. By contiguous allocation is meant the following. Instead of breaking the segment up into blocks of fixed size (pages), store the segment as one piece in a region of memory big enough to hold it. This scheme eliminates the waste of putting a small segment into a large page, since the region of memory holding the segment need be no bigger than that required to contain it. This is especially important for small segments. Thus contiguous allocation makes much more efficient use of memory.

What are the disadvantages of contiguous storage? Consider removing a segment from memory. The result is an area of free memory which is the size of the segment. The system must keep track of this and other existing free areas, so that when a new segment is to be put in memory an area the correct size can be found. There is also the possibility that for some particular segment being added there is no free area of a size to hold it. In this case it is necessary to rearrange the segments already in memory to make room. This is called compaction.

The advantages and disadvantages of contiguous allocation compared to paging are well known, and will not be detailed here further. The important observation to be made here is that because of certain features of the I/O task, the disadvantages of contiguous allocation are not as great as in the general case. The reason is that contiguous allocation is being proposed in addition to, not in replacement for paging. Thus whichever scheme is more advantageous can be used in any particular case. For example, the contiguous allocation scheme can be specialized for

small segments, for any segment near in size to a page or larger can be handled by paging. For another example, the overhead of the contiguous allocation scheme occurs at the time the segment moves in and out of memory, for at those times the area must be allocated and freed. Thus once a segment is in memory it is more efficient to keep it there for a long time, so that the overhead of bringing it in can be spread over many references to the segment. Happily, it is reasonable to use paging for segments which stay in memory a short time, since the waste from using a whole page to hold a small segment is proportional to the length of use. Thus the memory manager can pick the proper technique, based on the time limit supplied with the request to freeze the environment.

Another simplification results from the fact that, as noted above, segments for I/O need never grow. Growing a segment which is stored contiguously is expensive, for a whole new area must be found for it. This disadvantage of contiguous allocation is thus avoided.

The real memory needed to implement the contiguous allocation scheme can most conveniently be obtained from the blocks of memory used to hold pages. This source of storage would mean that allocation was done in several areas each the size of a page, rather than in one area, which would increase the waste area, but allow the amount of memory used for contiguous allocation to grow and shrink easily. This strategy is an example of tailoring the scheme for small segments; clearly using blocks of memory for allocation would not work if the scheme had to deal with segments larger than a page.

Note that if a contiguous allocation scheme were available for small segments, there are other uses which could be made of it. Chapter 6

discussed how storage must be used to hold the address mapping tables for the I/O process, and showed how the size of the tables might be reduced from those found in Multics. Clearly, contiguous allocation would be used to keep these small tables in core efficiently. There might well be other system tables which could be implemented as small segments (thus giving the user direct access) once the alternate allocation scheme were instituted. Thus the utility of the contiguous allocation scheme is not limited to I/O.

We turn now to the other objective of the chapter, which was to enlarge the class of devices for which the frozen environment is appropriate, while still fulfilling the goals of fair share resource distribution and compatibility with other system functions. As will be shown, the steps taken to reduce cost will assist in this endeavor too.

Fair Share Resource Distribution

In Chapter 3 the time limit on the frozen environment was proposed as a single solution which would achieve both goals at once. This section will attempt to broaden the class of acceptable devices by proposing a separate solution to each goal. We will first consider how to allocate resources in a fair fashion.

The effect of the time limit in resource allocation was to predict and control the extent of the committment represented by a request to freeze memory. Measuring the commitment in this way as a product of space and time, it should be clear that an efficient allocation strategy, which reduced the space required for a given request, can allow a proportionally longer time limit within the constraints of a fair share.

Thus the fixed time limit will continue to be the technique to enforce fair share allocation; the improvement will be the increased time limit allowed by more efficient allocation techniques.

What sorts of time limits might be reasonable? Again Multics will be used as an example. As part of typewriter I/O certain small areas of 16 words are allocated without any time limit at all. They are considered so small that they represent negligible storage consumption. The only limit is on the number of such areas which the I/O program may claim at once. This is a different interpretation of fair share. Earlier it was that resources would be made available some fraction of the time. Now it is that the user may have some small fraction of the resources all of the time.

Such a reinterpretation of the fair share criterion allows many more devices to use the frozen environment technique. Devices such as typewriters, which were formerly excluded, can now use the technique subject only to a restriction of the amount of resource consumed. The choice of whether to use the frozen environment then becomes a matter of economics. The user must compare the cost of keeping the resources in memory all the time with the cost on the other hand of external buffering plus fetching the resources as needed.

If the system cannot allow data to remain in memory for all time, then the technique will be insufficient for a continuously operating device. Clearly, the device can operate for no more than that fraction of the time the system will allow the resources to be frozen in memory. The sorts of devices which will operate under these conditions depends on details of the requirements. A typewriter could not operate continuously, but a typewriter which expected input only at certain times,

and which expected it within some time limit, could be made to work, since it is quite possible now to expect time limits on the order of minutes rather than seconds.

Without detailed information about the costs and capacities of a given system, it is impossible to predict the actual time limits which might be acceptable. Multics serves as an example, however, that even with today's costs, devices such as typewriters could be operated according to this technique if a compatible memory management scheme were devised. Thus, looking to tomorrow, with memory getting cheaper, it is reasonable to believe that proper memory management can essentially eliminate the fair-share problem.

Compatibility With Other System Functions

There remains the other goal, that of being compatible with other system functions. The example of a system function used earlier was reconfiguration of the real memory. This chapter has given another: compaction of the contiguous allocation area. In the original frozen environment scheme, the time limit, being enforced by the system, allowed reconfiguration processes to wait for the storage to become unfrozen. The maximum acceptable time limit was exactly the amount of time the reconfiguration process would be willing to wait. Again using Multics as an example, the maximum time reconfiguration would wait is probably a few seconds. The success we had in the last section stretching the limit out to minutes, or indefinitely, would thus vanish if time limits were used to insure the success of reconfiguration. It is thus necessary to abandon the time limit as a means of achieving this goal, and seek other

techniques. This section will introduce two techniques. The first moves the frozen area while the I/O is using it. The other observes that events such as reconfiguration are not frequent, so that the disruption caused by just doing the reconfiguration may be tolerated as a rarity. These alternatives will be considered in turn.

In order to see how it might be possible to move a piece of frozen virtual memory without disrupting I/O, consider the technique used in Multics to move pages which must remain in primary memory. This is a variantofa technique which was developed and described by Schell (36), who discussed reconfiguration in some detail. We begin with a review of address conversion. The tables which would be used to convert from virtual to real addresses were discussed in Chapter 2. The segment number was used as an index into the segment descriptor table, which gives, for each segment, the location of the page table for that segment. Each entry in the page table contains the current location of the given page; it also contains a bit, the modified bit, which is crucial to the reconfiguration algorithm. The bit is set on by the hardware whenever a reference which will write into the page is made. To move a page without disrupting the process which expects it to remain in memory, proceed as follows. First turn off the modified bit for the page, then make a copy of the page in the new location. The resulting copy may not be identical to the original if the original was modified during the copy. The modified bit can be inspected to see if the original has changed. If it has not, the address of the page which is contained in the page table entry can be changed to indicate the new copy, and processes referencing the page will continue smoothly using the new copy.

(Inspection of the modified bit and changing the address must be done as one indivisible step. This will be discussed below.) If the modified bit is on, the attempt to make a valid copy has failed. This failure causes no disruption to the operation of the system; it just means that another attempt must be made to copy the page.

Clearly, what is needed is some guarantee that if the copying operation is retried, it will eventually succeed. In the case of I/O, this is easy to guarantee, for I/O references a page at a regular rate, or at least at a maximum rate. Thus, in order that the copy be good, the copying operation must fit between two successive references to the page. Knowing how long it takes to copy a page allows us to calculate the maximum rate at which I/O can run if this scheme is to work. For example, on the current Multics, with the Honeywell 645 processor, a page can be copied in about 1.5 ms. This would mean an absolute maximum of an I/O reference each 1.5 ms. or 666 I/O references each second. If each reference transmitted 36 bits, the resulting bit rate would be about 24,000 bits per second.

Unless the copying operation is synchronized with the I/O references, there is no guarantee that the I/O may not just happen to reference the page during the copy. It is difficult to achieve synchronization; a better solution is to try the copying operation several times. If, for example, the actual allowed maximum I/O rate were less than one half the limit calculated above, then if during a copy an I/O reference occurred, it follows that before the next reference occurs there is bound to be time to fit in a second copying operation. Restricting the actual I/O transfer rate to one half the theoretic maximum is thus a simple solution to assure success of the copy on at most the second try.

Since the copying time is normally proportional to the size of the item to be copied, the cost reducing storage allocation technique developed in the last section can be used to advantage here. If, because of contiguous allocation, only the area itself need be moved, rather than the whole page which contains it, less time is required for the move. For example, if a 100 word area rather than a 1000 word page must be moved, the maximum acceptable I/O rate is increased tenfold.

What if a device is too fast for this technique? One obvious solution is to insist that such a device not run continuously. It would then be possible to use the alternate technique of waiting (with a fixed time limit) for the area to become unfrozen. Looking at practical devices, it is difficult to find a device which runs faster than the 24,000 bits per second estimated above for which the fixed time limit technique would not be applicable. Thus it might seem that these two techniques have covered all the situations, especially since the speed of processors is going up, which will increase the allowable device rate.

If there exist devices which run too fast and continuously, there is an alternative to the above technique, less elegant but perhaps useful, which is to turn off access to the page during the copying operation. This will have the effect of bringing the I/O process to an unexpected temporary halt as it attempts to reference the area. If this halt causes the irreversible loss of data, the technique is unacceptable. If the only result is that the device must stop, so that throughput is reduced, then the technique is probably usable, for events such as reconfiguration are normally infrequent so that the overall effect is minor.

It was noted above that the actions of checking the modified bit and changing the address in the page table entry must be done as one step. Clearly, if another process were to modify the area between the two steps, the modification would be lost. There are several ways these two steps could be done as one. The bit could be tested and the address stored using an interlocked read-rewrite memory cycle. Or an additional bit could be added to the page table entry which, when set, would cause the referencing processor to halt until the bit is turned off. This bit could be used to protect the two actions. The bit could also be used to halt the I/O during the actual copying operation, if that were to be done. Another way to make the actions indivisible is to stop all the other processors during the steps. This requires no special bit, but has a more widespread effect.

Additional complexity is introduced into these schemes if, as in Multics, an associative memory is used in the processor to remember recently used virtual-real mappings. The interested reader should be able to convince himself that by clearing the associative memory at the proper moments, the scheme will still work. For further details the reader may consult the thesis by Schell.

Summary

The purpose of this chapter has been to find an alternative to the fixed time limit technique which would allow a wider class of devices to use the frozen environment as an interface to the virtual memory. Such an alternative has been described. Central to its success was the efficient use of memory by means of some scheme such as contiguous

allocation. The goal of fair share resource distribution was still met by use of a time limit, with increased efficiency of memory usage lengthening the time limit to the extent that it might in practice be infinite. The goal of compatibility with the need of the system to rearrange virtual memory was met by abandoning the technique involving the time limit for a scheme in which the rearrangement occurs between successive memory references. The resulting variant of frozen environment is capable, under appropriate circumstances, of interfacing such devices as typewriters, which before were completly incompatible.

Chapter 8

Conclusion

The intent of this thesis has been to construct an I/O system which, in the context of a large virtual memory time sharing system, allows I/O control programs to be expressed naturally and to be executed efficiently. The purpose of this chapter is to review the system, and to consider the extent to which it has met its goals.

Chapters 3 through 7 have built up this I/O system as a series of additions to a basic framework presented in Chapter 2. The reader may feel a little uncertain as to the nature of the total system thus assembled, especially since several chapters have presented alternative techniques (such as Chapter 6, with four techniques for efficient use of processors) which lead to alternative forms of the total system. It thus may be helpful in review to summarize the system which results when these pieces are put together.

The arrangement of the physical modules which compose the system has been pictured in Figure 8-1. The devices are connected to the system via the device selector, which, in conjunction with the address mapping hardware, lets a processor refer to a device as if the device were a segment in a virtual address space. This device interface is the crux of the basic system developed in Chapter 2.

There are, from the user's viewpoint, two major additions to this basic system. One, a physical modification to the arrangement of the modules, is the several buffers pictured in Figure 6-1, which are used to cope with the timing characteristics of the devices, and which are used in conjunction with the technique of suspension and restoration

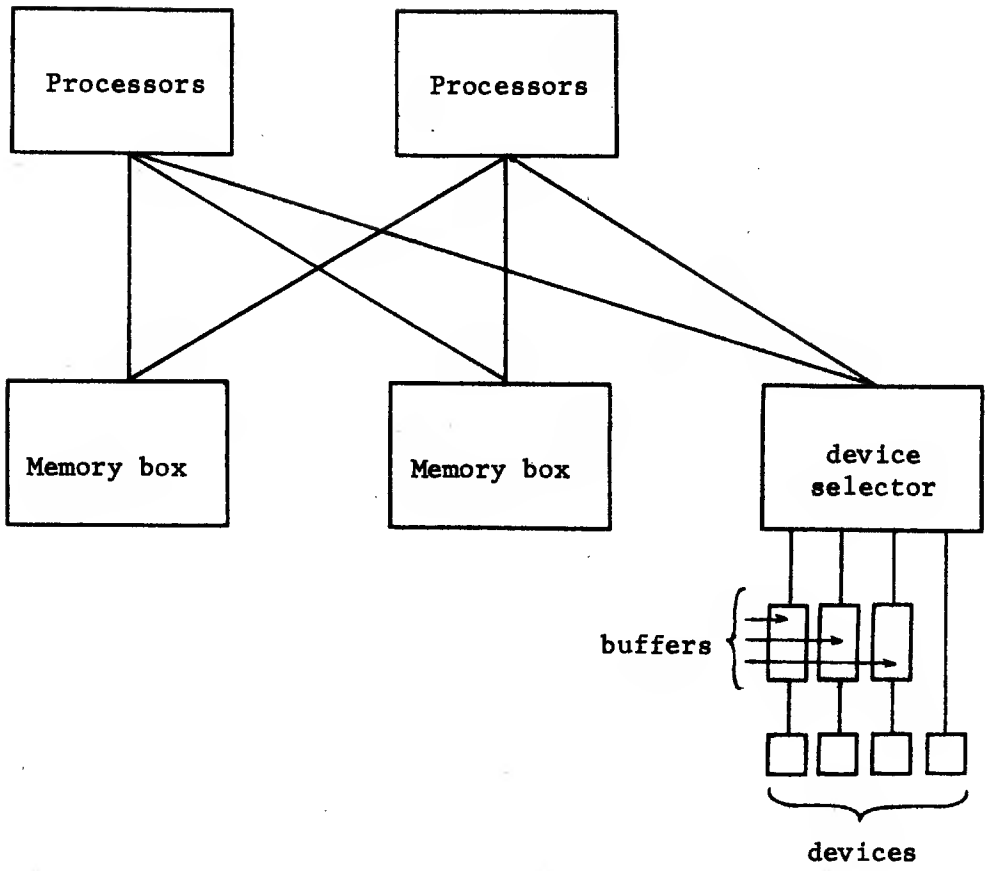

Figure 8-1: Final configuration of system modules.

to control the use of processors. The other addition to the basic system is the technique of the frozen environment, which does not involve modification to the hardware modules, but rather a modification to the software which comprises the virtual memory manager. This technique, like the buffer, is used to cope with the timing constraints of devices, and is used in conjunction with the technique of contiguous storage allocation to control the use of memory.

The fashion in which these two techniques are employed in the operation of a device depends on the characteristics of the device in question. There is a trade-off between the techniques, since both can be used to deal with timing delays, while each is useful in eliminating some other problems. For example, in the case of very slow or very fast devices, it will be possible to dispense with the use of the buffers altogether, which is desirable when possible, because of the complexities of buffers. It is perhaps unfortunate that because of the choice between these two techniques no single thumbnail sketch can be constructed showing the operation of a device. It is unfortunate but it is also crucial that this choice exist, in order to cope with the wide variation in device characteristics. For example, the range of transmission rates found in devices varies by more than six orders of magnitude. It should not be surprising that any architecture which can deal with this variability must contain some choices.

As the various chapters have added pieces to the basic system of Chapter 2, they have also added new lines to the interface between the device and the device selector. Appendix B has been included for the benefit of those readers who would like a review of the interface in its

final form.

One module which was omitted from Figure 8-1 was the specialized processor, or SP, of Chapter 6. The reader may remember that the SP was proposed as an alternative to the buffer, in order to deal with certain special problems. Thus it has been assumed that for the system summarized here SPs are unnecessary. Another possible system which could be built out of the pieces described in the various chapters would be a configuration which used SPs to control the cost of processors, and which dispensed with the use of buffers completely by using the frozen environment to deal with all questions of timing. Such an alternative might be appropriate in certain cases.

The above summary of the system reflects the user's point of view. Another form of a summary is a list of the particular modules in the operating system which must be modified or created so that the I/O works properly.

- A contiguous storage allocator must be created to provide storage at acceptable cost for the small segments typically used in I/O.
- The virtual memory manager must be modified so that it accepts requests to freeze and unfreeze the environment. This will require that the manager use the contiguous storage allocator, and also that it interface to the modules responsible for managing resources other than memory.
- The processor scheduler must be modified so that in response to the need processor signal the relevant process is scheduled.
- A device manager must be created, which allocates devices to

particular processes, and which creates the segment representing the device in the virtual address space of the process.

- The memory reconfiguration routine must be modified so that it can deal with frozen memory.

Yet another way to summarize this system is to remember the two goals propsed in Chapter 1, which were that the user have direct access to his device, and that he be able to construct his I/O control program in a natural manner, and to note which features of the system are important in achieving these goals.

The first goal was that the user be allowed direct access to his device, or put another way, that it not be necessary for some system program to interpret the user's I/O control program for him. Three features of the system contribute to this goal. First, the representation of the device as a segment provides a means of controlling the user's access to each device individually. Otherwise some special registers would be needed to control which user accesses which device. Secondly, the fact that the I/O control program executes in the environment of the user means that the program is automatically constrained by the protection controls on the environment. Multics is an example of the alternative; the channels which perform the I/O in Multics do not have address mapping ability, so they must operate in the environment of real rather than virtual addresses. Since this environment provides no memory protection, the user cannot write programs for the channels. The third point is related to this last one; it is presumed that the cost of activating the user's environment is sufficiently small that it is economically reasonable to provide this environment each time the I/O control program runs. In other words

it must be cheap to start a process running. These three features: representation of the device as a segment, running the I/O control program in the user's environment, and keeping the I/O process cheap to bring into execution, are the crucial features in giving the user direct access to his device.

The other goal was that the user be able to construct his I/O control program simply and naturally. Two features of the system contribute to achieving this goal. The first is the elimination of the interrupt from the environment of the user, and the elimination of the so-called "interrupt handler structure" of the I/O control program. Chapter 2 argued at some length that the natural form of the I/O control program was sequential, rather than being structured by interrupts. The other feature which contributes to the goal of programming simplicity was the ability to write the I/O control program using the language of the central processor, or in fact using a high level language.

This thesis has proposed one particular I/O architecture. It should be clear to the reader that there is nothing unique about the details of the architecture. Indeed, the thesis itself has pointed out certain alternatives to the scheme. The role of this particular proposal is, first, to serve as an existence proof that in the context of virtual memory it is possible to construct an I/O system which allows the user to program his device directly in a natural and efficient manner.

The other role of this proposed architecture is to clarify the various interactions which exist among the features of the I/O system. It is clear that to build a successful I/O architecture, one must solve

several problems. One must deal with issues of protection, efficiency, program structure, timing and so on. Further, it is clear that each feature of the I/O system will have an effect on several of these problems. The resulting interaction between the features means that one cannot consider each feature in isolation, for the success of a feature depends on the fashion in which it meshes with other features. This thesis has tried to find an order for considering the problems of I/O which reduces the degree of interaction between the various features, first considering issues of program structure, then considering timing and efficiency. Thus again the architecture here proposed serves as an existence proof that there exists an orderly procedure for designing an I/O system.

Future Research

An obvious question which must be asked about any research such as this is where to go from here. The purpose of this section is to identify various areas in which further research would be appropriate and fruitful.

It would be most valuable to test out the I/O system proposed in this thesis by implementing it. Only in this way can the practicality of the system be proven. More importantly, only by implementing the system will it be possible to determine how users will take advantage of it. This thesis has not proposed specific I/O strategies which the user might employ; rather it has built a framework within which the user is given the freedom to construct whatever mechanisms he needs. By observing the mechanisms which he actually builds it may be possible to discover new tools or modifications to existing facilities which the system ought to provide for the user or simplifications which would be acceptable.

There are several parts of this I/O system which would have to be specified much more completely if the system were to be implemented. For example, several design decisions must be made about the device selector. The selector must be implemented in such a way that requests from one processor do not interfere with or delay requests from other processors, because a request to read or write data may be pending for a considerable time. The selector must operate properly if a process with a pending read or write is suspended and restored. And the behavior of the selector must be specified for the case where more than one process tries to reference the same device. There are also detailed questions about the interface between the device and selector, such as what sequence of signals occur if a pending operation is aborted and another started. These sorts of questions have not been considered in the thesis, for they do not contribute to the understanding of the basic structure being developed, but clearly in a practical case they would be important.

This thesis has not discussed certain auxiliary modules needed as part of the I/O system. For example, the system must contain a module which is called at the time a device is assigned to a process, whose function is to regulate the utilization of devices. It must make certain that one process does not hog devices to the detriment of the others, it must confirm the user's authorization to use the device, and it must implement any charges which the system imposes for the use of devices.

It is common to alter the number and arrangement of devices attached to the system, so it is important that there be an orderly way for the I/O system to determine which devices are connected. It might be useful or necessary to add a new line to the device interface by

means of which the processor may check the existence and identity of the device. The module described in the previous paragraph, which regulates device utilization, must have access to this information in order to make proper allocation decisions. Hopefully, it will be possible to change the arrangement of devices while the system is running.

One aspect of I/O which could be considered in greater detail is what features should be present in a high-level language which is to be used for I/O. This thesis has discussed to a certain extent the semantics of I/O, and has talked about program structure, but there remain such questions as the language representation of the segment which is the device, the syntax for error recovery and process synchronization, and the form of the data transfer operation. In general it would be worth-while to catalog the features which must be present in a language so that it can perform I/O.

The thesis has mentioned that conversion from virtual to real addresses is usually expedited by means of an associative memory which remembers virtual-real relationships. If a specialized processor, or SP is used for any I/O, then it is necessary to decide whether the SP should have an associative memory, and how that memory should be structured. For example, could parts of the associative memory be shared among several SPs to reduce the cost? For one view of this problem the reader may see the thesis by Smith (37).

This thesis has restricted itself to considering I/O performed by the user, and has not discussed I/O performed by the system, for example disk or drum I/O to support paging. There is no reason, however, why the device interface described here cannot be used by the system as well,

and this is clearly desirable, as it avoids the need for a separate I/O structure for the system. The system would use this I/O structure in a slightly different manner than the user, however, and this thesis has not considered these differences. For example, in what address space does the system refer to its device? What process structure does the system use to handle errors? What timing problems does the system have and how does it cope with them?

Chapter 6 discussed the possibility that execution delays might be introduced in the I/O control program because of processor scheduling, and it suggested that buffering might be used to deal with these delays. Buffering may be inappropriate if, for example, a computation must be performed promptly on an incoming item, for buffers do not eliminate delays. An alternative solution is to modify the process scheduler so that it is able to provide a guaranteed upper limit on the time from the arrival of the need processor signal until the process is running. The thesis has not discussed such a scheduler, and it would be an interesting project to show that one could be integrated into this scheme. One scheduler which the author believes to be suitable is described by Fiala (19) and by Strollo, Tomlinson and Fiala (38).

A problem which is currently under study in the computer industry is how to build a multiple processor which uses cache memories. The cache memory, a small, fast memory used to hold recently referenced pages in a quickly accessible fashion, is a part of the processor. Thus if two processors reference the same page they will each get a separate copy in their their own cache, and if one or both write in the page, they will create two inconsistent copies. Thus any multiple processor cache

system must have some inter-processor communication scheme to insure that the copy in each cache is identical. I/O can interact with the use of caches because one of the two processors referencing a page may be an SP, or specialized processor. While an SP might not have a cache, it must have all the mechanisms needed to maintain consistent copies, for it may modify a page which is currently in the cache of some other processor. The addition of this mechanism to the SP may run counter to the desire to keep the SP as simple as possible. Another difficulty arises if there are a large number of SPs, as there might be if SPs are indeed very cheap. A large number of SPs might render impractical any co-ordination scheme which required each processor to be connected to every other.

While in these fashions I/O may make the use of caches more difficult, I/O also provides a simplification if the frozen environment scheme is being used, for the system can always tell which pages may be referenced by an SP. This knowledge may represent a way to special-case the problem of SPs and caches.

Appendix A

Details of Buffer Algorithms

Chapter 4 describes a particular buffer algorithm which had as a goal that the interface which the buffer presents to the selector should be identical in behavior to the interface provided by the device itself, and similarly that the interface which the buffer presents to the device be identical to that from the selector. This appendix presents the details of these algorithms for the benefit of the reader who wishes to confirm that the algorithms can be constructed, and to give an idea of their complexity.

The device-selector interface is reviewed in detail in Appendix B, where the function of all the lines is discussed. The algorithm involves the following lines from the interface:

- Command lines, from selector to device -- the two commands used in these algorithms are read-data and write-data.
- Ready acknowledge pair to control command lines -- (c-rdy and c-ack).
- Data lines, in either direction -- (d).
- Read and write ready-acknowledge pairs, to control data lines -- (r-rdy, r-ack, w-rdy, and w-ack).
- Reverse write ready line, from device to selector -- (rev-w-rdy); this was introduced in Chapter 5.
- Read operation required (ror) and write operation required (wor) from device to selector.

In review, the procedure for transferring an item across the interface consists of two parts: first the command, from the selector

to the device, and second the item itself, on the data lines. The transfer of the command and of the item are each under control of a pair of lines called ready and acknowledge. When the command or item has been placed on the appropriate lines, a signal will be sent from sender to receiver on the appropriate ready line, indicating that the information may be read. When the item has been successfully read, the receiver will signal back to the sender on the associated acknowledge line.

The first algorithm to be presented will read data from device to selector. As stated in Chapter 4, it was as follows:

> When empty buffer receives read data command from selector, acknowledge it and pass it on to the device.
>
> After handling the read command, wait for device to send data back. When data arrives, acknowledge it, and pass it on to the selector.
>
> When empty buffer receives ror signal from device, fabricate a read data command and send it to device. Send ror to selector. When data arrives from device and read data command arrives from selector, pass data to selector.

In order to specify in detail all the various sequences of signals which may arrive at the two interfaces to the buffer, this algorithm is presented in Figure A-1 as a state transition diagram. In this diagram, states 1-4 represent the first two parts of the algorithm; states 5-9 deal with the ror signal.

Two algorithms were presented for the writing of data, one with and one without the write operation required line (wor). The one not involving the wor line, being simpler, has been preferred throughout the thesis. Both will be presented here to show the relative complexity.

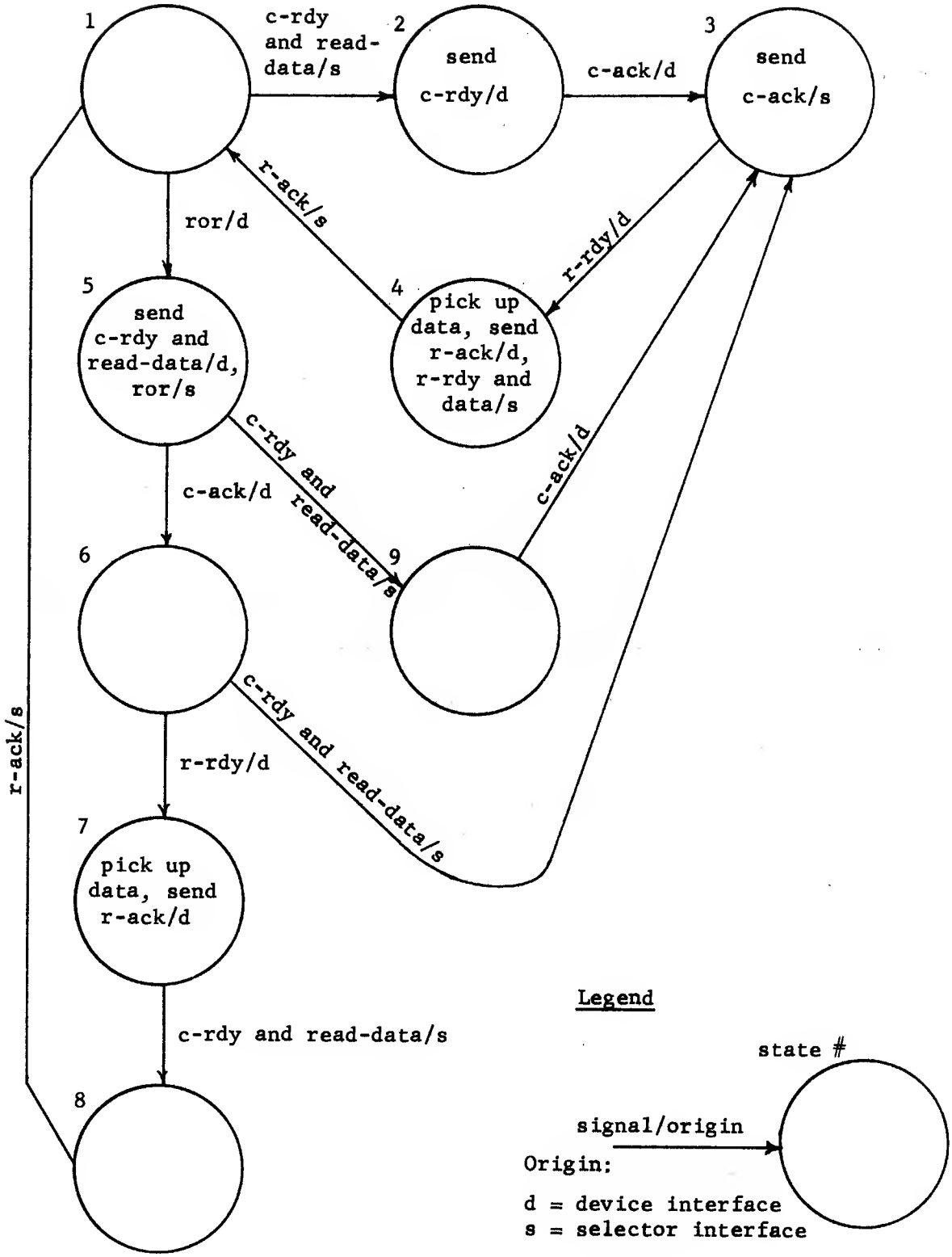

Figure A-1: Read data algorithm for buffer.

The algorithm not involving the wor line is as follows:

> Whenever the buffer contains a data item from the selector, send a write data command to device, and an acknowledgement, send data to device. When a write data command arrives from selector, acknowledge, to wait until buffer is empty. Then wait for data from selector, pickup data and acknowledge it.

This is pictured as a state transition diagram in Figure A-2. Figure A-3 presents the variant which contains the wor line. As the diagrams show, the use of the wor line adds considerable complexity to the buffer's algorithm.

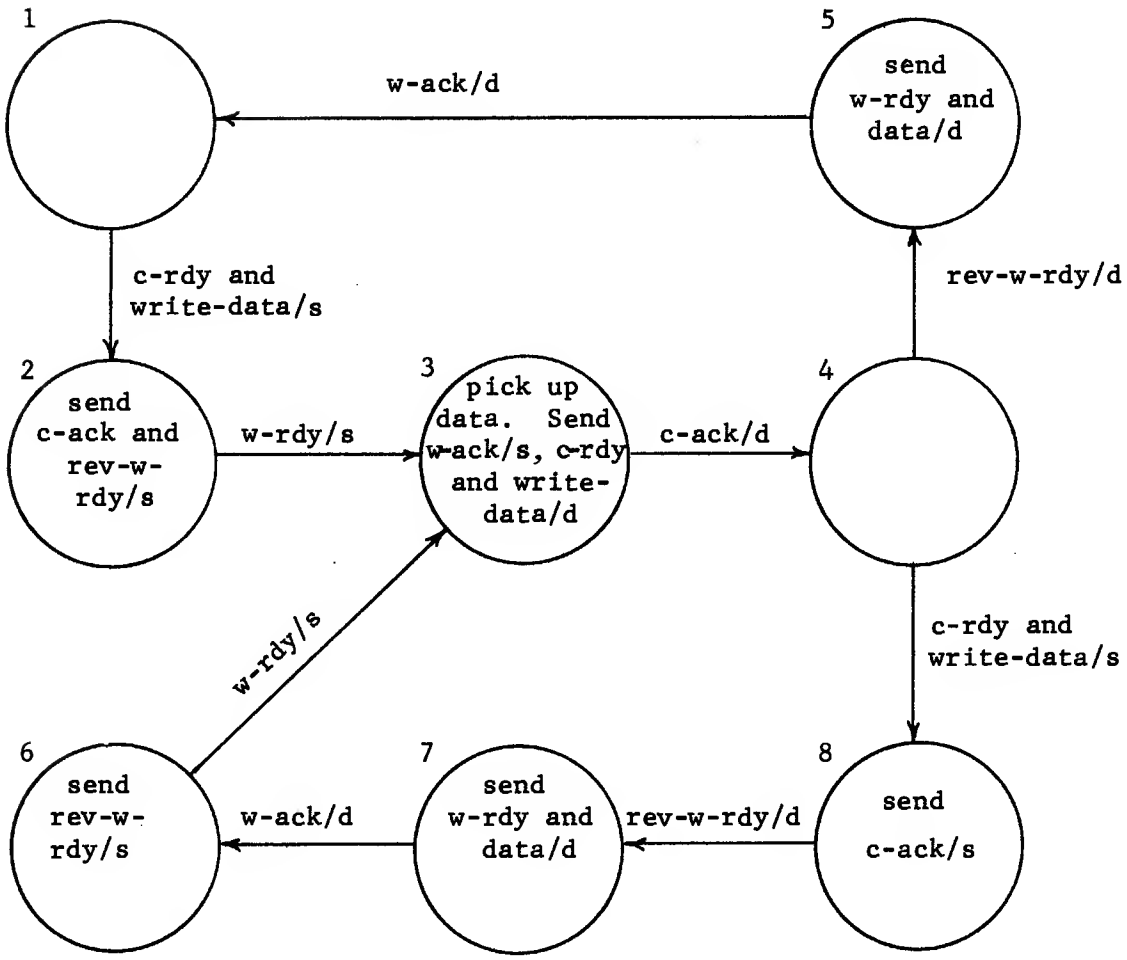

Figure A-2: Write data algorithm for buffer without wor line.

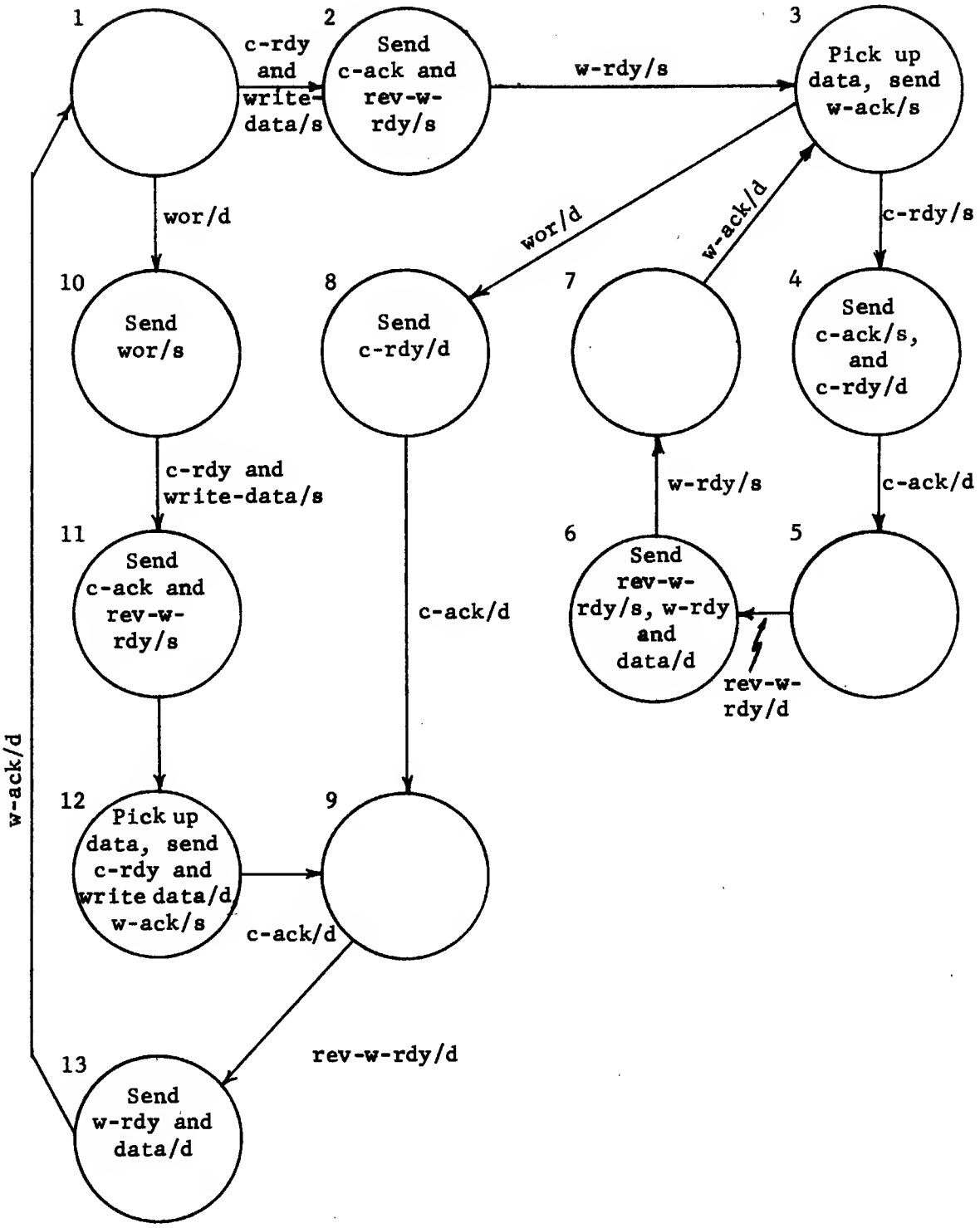

Figure A-3: Write data algorithm for buffer with wor line.

Appendix B

Review of Interface Between Device and Device Selector

The interface between device and device selector has been designed to allow asynchronous interchange of data, state information, record numbers, and potentially other sorts of values. Modifications to the interface have allowed it to operate successfully in conjunction with buffering, with multiplexing of various sorts, and with processor scheduling.

Because the interface is asynchronous, control lines are needed to co-ordinate the transfer of information. In particular, two lines, called ready and acknowledge, are used for any set of information lines. The ready line runs from the sender to the receiver of the information, and a signal over it means that the information lines are now carrying a valid set of signals and may be read. The acknowledge line runs in the reverse direction, from the receiver to the sender of the information, and a signal over this line means that the information has been received, and that the sender of the information need no longer hold the information lines in a valid state.

The various lines in the interface are diagrammed in Figure B-1. The following is a description of the function of each line.

Command - The steps necessary to move an item of information across the interface consist of two parts: the first a command from selector to device declaring what is to be transferred and in which direction, the second the item itself. These lines carry the command. It is presumed that there are sufficient lines to distinguish the various commands.

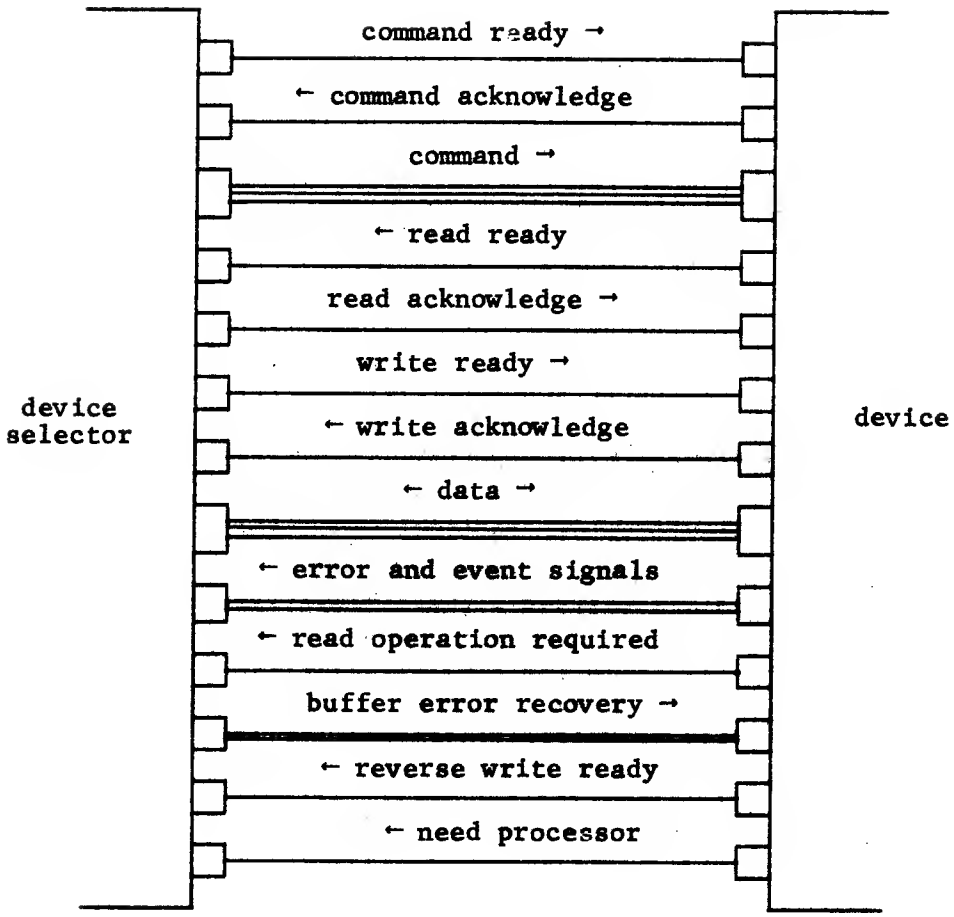

Figure B-1: Complete device-selector interface.

Command ready and command acknowledge - these two lines control the flow of information across the command lines, in the fashion described above.

Data - the second half of each transaction at the interface is the transfer of the item specified by the command. These lines carry the item itself. Whether the item is data, state word, or record number; and whether the item is read or written is specified by the command. The number of lines in this connection has not been specified as part of the thesis , but unless the interface is modified there must be sufficient lines to hold the state word and the record number information.

Read ready and read acknowledge - these lines control flow of information in the data lines in the case in which the information moves from device to device selector.

Write ready and write acknowledge - these lines control the flow of information on the data lines in the case in which the information flows from selector to device.

Reverse write ready - in order that certain kinds of multiplexing work properly, it is necessary that the device itself generate a signal whenever it is ready to perform the data transfer. For reading, the read ready line serves this function; the reverse write ready is provided in the case of writing. The meaning of the signal to a multiplexor is that it should now assign itself to this device. This line is not strictly a ready line, for it does not control information transfer as a ready line does, but the buffer algorithms of Appendix A

have been adjusted so that the write ready signal is generated only in response to a reverse write ready.

Error and event signals - these are lines which the device uses to report errors and events as discussed in Chapter 2. Errors and events are distinguished in that an error signal reports a synchronous occurence and results in an error handler being run on the I/O process, whereas an event signal reports an asynchronous event and results in the scheduling of an event process.

Read operation required - this line runs from device to buffer, and is used during reading, to force the buffer to accept an item from the device if the processor lags behind. See Appendix A for the specific sequence of signals.

Buffer error recovery - these lines run from the processor through the buffers, and serve to restore the buffer to a known state after an error. Two specific recoveries were discussed in Chapter 4: discarding the buffer contents and reversing the direction of flow from writing to reading.

Need processor - this signal is received by the processor, and is generated by a device or by a buffer. Its meaning is that the I/O process in charge of the service should be scheduled.

Comparison with Other I/O Interfaces

A discussion of two other I/O interfaces will give some further insight into the operation of this interface, and will at the same time show that this interface is not greatly different from interfaces in use today.

The interface which is used on Multics for control of devices other than communication devices is the "Common Peripheral Interface" (25) which is standard over much of the Honeywell line. In major respects it is similar to the one proposed here. All information is transferred asynchronously, using control lines similar to ready and acknowledge. Also, one set of lines is used both for data and status, the uses being distinguished by a command. The interface differs in that there are separate lines in each direction, and the data lines to the device also carry the commands. A special control line is used to signal the presence of a command on these lines.

There are two important differences between this interface and the one proposed in this thesis. First, the Honeywell interface has four lines running from the device, which at all times indicates the state of the device. Thus it is always possible to test the state without sending a command and receiving a state word across the interface. This avoids the complex interaction which results from needing to test the state in the middle of some other interface transaction. Since additional state information can be obtained by sending a command across the interface, these four lines could be viewed as a cross between state information and the error and event lines. The Honeywell interface does contain a separate line to report asynchronous events.

The other important distinction between the interfaces is that the Honeywell interface allows a command to trigger not one but a number of data transfers. The advantage of this is that it increases throughput across the interface by eliminating the repeated command transfer. This form of the command is consistent with the view that a channel executes

a single instruction which results in a sequence of data transfers. The one instruction would then trigger the one command. In the system of this thesis, in which each data transfer is performed by a separate machine instruction, it is more reasonable to imagine the command to be generated and sent anew as part of each transfer. However, one could devise a special line to achieve the same effect as the Honeywell interface: a line from selector to device whose meaning is "use again the same command as last time".

The IBM System/360 and System/370 have a standard interface between channel and device control unit which serves the same function as the interfaces so far discussed (26). It is again similar, with asynchronous transfers over one set of data lines in each direction regulated by control lines similar to ready and acknowledge. Like the Honeywell interface, the IBM interface allows one command to trigger not one but a sequence of transfers. The IBM interface differs in that the sequence of control signals across the interface is much more complicated, to some extent because the channel may be shared among several devices, so that the interface control signals must also select the proper device. An interesting question is whether the protocol used to select the device could be used as a means of implementing the device selector of this thesis.

Bibliography

(1) Bolt Beranek and Newman, Inc., Specifications for the Interconnection of a Host and an IMP, Report 1822, Cambridge, Mass.

(2) Boulton, P.I.P., and P. Reid,"A Process-Control Language," IEEE Trans. Computers, Vol C-18,11, Nov. 1969.

(3) Chang, W., "Computer Channel Interference Analysis," IBM Systems Journal, Vol 4,2, 1965.

(4) Chu, W.W., "Buffer Behavior for Batch Poisson Arrivals and Single Constant Output," IEEE Trans. Communication Technology, Vol COM-18,5 Oct. 1970.

(5) Chu, W.W., "A Study of Asynchronous Time Division Multiplexing for Time-sharing Computer Systems," Proc. AFIPS 1969 FJCC, Vol. 35 AFIPS Press, Montvale, N.J.

(6) Chu, W.W., "Buffer Behavior for Poisson Arrivals and Multiple Synchronous Constant Output", IEEE Trans. Computers, Vol C-19,6 June 1970.

(7) Chu, W.W., "Design Considerations of Statistical Multiplexors," ACM Symposium on Problems in Optimization of Data Communication Systems, Pine Mtn, Georgia, Oct. 1969.

(8) Corbató, F.J., J.H. Saltzer, and C.T. Clingen, "Multics -- The First Seven Years," Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale N.J.

(9) Corbató, F.J., and Vyssotsky, V.A., "Introduction and Overview of the Multics System," Proc. AFIPS 1965 FJCC, Spartan Books, Washington, D C.

(10) Cosserat, D.C., "A Capability Oriented Multi-processor System for Real-time Applications," International Conference on Computer Communication, Washington, D C, 1972.

(11) Digital Equipment Corporation, PDP-11 Processor Handbook, Maynard Mass, 1971.

(12) Digital Equipment Corporation, PDP-11 Peripherals and Interfacing Handbook, Maynard, Mass., 1971.

(13) Delgalvis, I, and J.P. Bricault, "An Analysis of a Request Queued Buffer Pool," IBM Systems Journal, Vol 5,3, 1966.

(14) Delgalvis, I, and G. Davison, "Storage Requirements for a Data Exchange," IBM Systems Journal, Vol 3,1, 1964

(15) Dijkstra, E.W., "The Structure of the 'THE'-Multiprogramming System," Comm ACM Vol 11, 5, May 1968.

(16) Dor, N.M., "Guide to the Length of Buffer Storage Required for Random (Poisson) Input and Constant Output Rates," IEEE Trans. Electronic Computers, Vol EC-16, Oct. 1967

(17) England, D.M., "Operating System of System 250," International Switching Symposium, MIT Cambridge Mass, 1972

(18) Feiertag, R.J., and E.I. Organick, "The Multics Input/Output System," ACM Third Symposium on Operating Systems Principles, Oct 1971.

(19) Fiala, E., Scheduling of Real-time Processes in a Time-shared Environment, MS Thesis, Dept. Electrical Engineering, MIT, 1968.

(20) Gaver, D.P., and Lewis, P.A.W., Probability Models For Buffer Storage Allocation Problems, IBM Research Report No. RC 2590, Aug 1969.

(21) Gertler, J., "High-level Programming for Process Control", Computer Journal, Vol 13,1, Feb. 1970.

(22) Halton, D., "Hardware of the System 250 for Communications Control," International Switching Symposium, MIT, Cambridge, Mass. 1972.

(23) Hatch, T.F. Jr., and J.B. Geyer, "Hardware/software Interaction on the Honeywell Model 8200", Proc. AFIPS 1968 FJCC, Vol 33, AFIPS Press, Montvale, N.J.

(24) Held, M., and R. Carp, "Dynamic Programming and Sequencing Problems" J. Soc. for Industrial and Applied Mathematics, Vol 10,1, 1962.

(25) Honeywell Information Systems, Inc, Product Performance Specification Common Peripheral Interface, No. 43A130524

(26) IBM, System/360 and System/370 I/O Interface- Channel to Control Unit (Original Equipment Manufacturers' Information), GA22-6974-0

(27) Manchester, G.K., "Production and Stabilization of Real-time Task Schedules," JACM Vol 14,3 July 1967.

(28) McKenzie, A., Host/Host Protocol for the ARPA Network, Network Information Center Doc. 8246, Augmentation Research Center, Stanford Research Institute, Menlo Park, Cal.

(29) McQuillan, J.M., and others, "Improvements in the Design and Performance of the ARPA Network," Proc. AFIPS 1972 FJCC, Vol 41, AFIPS Press Montvale, N.J.

(30) Muntz, R.R., and E.G. Coffman, Jr., "Preemptive Scheduling of Real-time Tasks on Multiprocessor Systems," J ACM Vol 17,2, April, 1970.

(31) Organick, E.I., The Multics System: An Examination of its Structure, MIT Press, 1972.

(32) Ossanna, J.F. and others, " Communication and Input/Output Switching in a Multiplex Computing System," Proc AFIPS 1965 FJCC, Spartan Books Washington DC.

(33) Plessey Telecommunications Research, Ltd., Architectural Features of System 250 .

(34) Roberts, L.G. and B.D. Wessler, "Computer Network Development to Achieve Resource Sharing", Proc AFIPS 1970 SJCC, AFIPS Press, Montvale, N.J.

(35) Saltzer, J.H., Traffic Control in a Multiplexed Computer System, ScD Thesis, MIT Dept. Electrical Engineering, 1966. Also as Project MAC report TR-30.

(36) Schell, R.R., Dynamic Reconfiguration in a Modular Computer System, PhD Thesis, MIT Dept. Electrical Engineering, 1971. Also as Project Mac report TR-86.

(37) Smith, A.A., Input/Output in Time-shared, Segmented, Multiprocessor Systems, MS Thesis, MIT Dept. Electrical Engineering, 1966. Also as Project MAC report TR-28.

(38) Strollo, T.R., R.S. Tomlinson, and E.R. Fiala, "A Time-shared I/O Processor for Real-time Hybrid Computation," Proc. AFIPS 1969 FJCC AFIPS Press, Montvale, N.J.

(39) Telnet Protocol Specification, Network Information Center Doc. 15372 Augmentation Research Center, Stanford Research Institute, Menlo Park, Cal.

(40) Wolman, E., "A Fixed Optimum Cell-size for Records of Various Lengths" JACM Vol 12,1, Jan. 1965.

(41) Wirth, N., "On Multiprogramming, Machine Coding, And Computer Organization," Comm. ACM, Vol 12,9, Sept 1969.

Biographical Note

David Dana Clark was born on April 7, 1944. He grew up in St. Louis, Missouri, where he attended John Burroughs High School. He then attended Swarthmore College, majoring in Electrical Engineering. He was awarded the McCabe Engineering Award as the outstanding engineering student in his class. In 1966 he received the degree of BSEE with distinction.

He attended graduate school in Computer Science at MIT starting in 1966, receiving the MS and EE degrees in September, 1968. Since 1967 he has been associated with Project MAC, where his principal interest has been research on the Multics system. He has also worked in the area of programming linguistics; one project was the design of a high-level language for operating system implementation.

He was a teaching assistant in introductory circuit theory courses at MIT; he also taught a semester introductory course on computers at Wellesley College. As part of this latter project, he participated in the writing of a manual for the language PL, which has since been used at MIT.

He is a member of Sigma Xi, Sigma Tau, the IEEE, and the ACM.

His Master's thesis was: A Reductions Analysis System for Parsing PL/I

He has written:

> The Classroom Information and Computing Service, Project MAC report TR-80, January 1971 (with M.D. Schroeder, R.M. Graham, and J.H. Saltzer).
>
> The Programming Language PL, MIT Dept. Electrical Engineering, 1969 (with A.L. Anger, A.A. Bushkin, and J.R. Coffman).

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 2/29/96

Report # LCS-TR-117

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 192 (198-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): FOLLOWS LAST PAGE (191)

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-192) UNMED TITLE PAGE, 2-191, UNMED BLANK
(193-198) SCANCONTROL, DOD(2), TRGT'S(3)

Scanning Agent Signoff:
Date Received: 2/29/96 Date Scanned: 3/1/96 Date Returned: 3/12/96

Scanning Agent Signature: Michael N. Cook

BIBLIOGRAPHIC DATA SHEET	1. Report No. MAC TR-117	2.	3. Recipient's Accession No.
4. Title and Subtitle An Input/Output Architecture for Virtual Memory Computer Systems			5. Report Date: Issued January 1974 6.
7. Author(s) David D. Clark			8. Performing Organization Rept. No. MAC TR-117
9. Performing Organization Name and Address PROJECT MAC: MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139			10. Project/Task/Work Unit No. 11. Contract/Grant No. N00014-70-A-0362-0006
12. Sponsoring Organization Name and Address Office of Naval Research Department of the Navy Information Systems Program Arlington, Va 22217			13. Type of Report & Period Covered: Interim Scientific Report 14.

15. Supplementary Notes

Ph.D. Thesis, Department of Electrical Engineering

16. Abstracts: In many large systems, user I/O must be performed for the user by the system, in order to assure such system goals as security, response, and efficiency. However, reduced overhead and increased flexibility would result if the user could perform his I/O directly. This thesis presents a design for an I/O subsystem architecture which, in the context of a segmented, paged, time-shared computer system, allows the user direct access to I/O devices. Some conclusions of this thesis are:1) that in order to provide a coherent program structure, I/O operations should be contained in a separate I/O process, 2) that to allow the user to refer to his devices in a simple fashion while protecting his devices from other users, the I/O device should be represented to the user as a segment, 3) that the virtual memory can meet the timing needs of the I/O system without compromising its own functions by the use of time limits on the duration of the I/O operations, and 4) that interrupts should not be part of the user environment, but should be hidden from the programmer, so that the I/O program he constructs is sequential rather than interrupt driven in structure.

17. Key Words and Document Analysis. 17a. Descriptors

Computer Operating Systems

Input/Output

Virtual Memory

Time-Sharing

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited Distribution Write Project MAC Publications	UNCLASSIFIED	192
	20. Security Class (This Page) UNCLASSIFIED	22. Price

INSTRUCTIONS FOR COMPLETING FORM NTIS-35 (10-70) (Bibliographic Data Sheet based on COSATI Guidelines to Format Standards for Scientific and Technical Reports Prepared by or for the Federal Government, PB-180 600).

1. **Report Number.** Each individually bound report shall carry a unique alphanumeric designation selected by the performing organization or provided by the sponsoring organization. Use uppercase letters and Arabic numerals only. Examples FASEB-NS-87 and FAA-RD-68-09.

2. Leave blank.

3. **Recipient's Accession Number.** Reserved for use by each report recipient.

4. **Title and Subtitle.** Title should indicate clearly and briefly the subject coverage of the report, and be displayed prominently. Set subtitle, if used, in smaller type or otherwise subordinate it to main title. When a report is prepared in more than one volume, repeat the primary title, add volume number and include subtitle for the specific volume.

5. **Report Date.** Each report shall carry a date indicating at least month and year. Indicate the basis on which it was selected (e.g., date of issue, date of approval, date of preparation.

6. **Performing Organization Code.** Leave blank.

7. **Author(s).** Give name(s) in conventional order (e.g., John R. Doe, or J.Robert Doe). List author's affiliation if it differs from the performing organization.

8. **Performing Organization Report Number.** Insert if performing organization wishes to assign this number.

9. **Performing Organization Name and Address.** Give name, street, city, state, and zip code. List no more than two levels of an organizational hierarchy. Display the name of the organization exactly as it should appear in Government indexes such as **USGRDR-I.**

10. **Project/Task/Work Unit Number.** Use the project, task and work unit numbers under which the report was prepared.

11. **Contract/Grant Number.** Insert contract or grant number under which report was prepared.

12. **Sponsoring Agency Name and Address.** Include zip code.

13. **Type of Report and Period Covered.** Indicate interim, final, etc., and, if applicable, dates covered.

14. **Sponsoring Agency Code.** Leave blank.

15. **Supplementary Notes.** Enter information not included elsewhere but useful, such as: Prepared in cooperation with . . . Translation of . . . Presented at conference of . . . To be published in . . . Supersedes . . . Supplements . . .

16. **Abstract.** Include a brief (200 words or less) factual summary of the most significant information contained in the report. If the report contains a significant bibliography or literature survey, mention it here.

17. **Key Words and Document Analysis. (a). Descriptors.** Select from the Thesaurus of Engineering and Scientific Terms the proper authorized terms that identify the major concept of the research and are sufficiently specific and precise to be used as index entries for cataloging.
(b). Identifiers and Open-Ended Terms. Use identifiers for project names, code names, equipment designators, etc. Use open-ended terms written in descriptor form for those subjects for which no descriptor exists.
(c). COSATI Field/Group. Field and Group assignments are to be taken from the 1965 COSATI Subject Category List. Since the majority of documents are multidisciplinary in nature, the primary Field/Group assignment(s) will be the specific discipline, area of human endeavor, or type of physical object. The application(s) will be cross-referenced with secondary Field/Group assignments that will follow the primary posting(s).

18. **Distribution Statement.** Denote releasability to the public or limitation for reasons other than security for example "Release unlimited". Cite any availability to the public, with address and price.

19 & 20. **Security Classification.** Do not submit classified reports to the National Technical

21. **Number of Pages.** Insert the total number of pages, including this one and unnumbered pages, but excluding distribution list, if any.

22. **Price.** Insert the price set by the National Technical Information Service or the Government Printing Office, if known.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**